Genetic Algorithms for Optical Character Recognition

by

Joseph John Svitak, Jr.

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2008

This manuscript has been read and accepted for the
Graduate Faculty in Computer Science in satisfaction of the
dissertation requirement for the degree of Doctor of Philosophy.

Dr. Robert Goldberg

Date — Chair of Examining Committee

Dr. Ted Brown

Date — Executive Officer

Dr. Jinlin Chen, Queens College

Dr. David B. Fogel, Natural Selection, Inc.

Dr. K. L. Kwok, Queens College

Dr. Jerry Waxman, Queens College

Supervisory Committee

The City University of New York

Abstract

Genetic Algorithms for Optical Character Recognition

by

Joseph John Svitak, Jr.

Adviser: Professor Robert Goldberg

This thesis addresses the application of genetic algorithms to optical character recognition (OCR). The first problem considered is recognizing characters. Agents (finite state machines) evolve that dedicate themselves to particular pathway segments of a noisy (possibly handwritten) character. The fitness of an agent is the amount of the path consumed by the agent. Collectively, these agents have the means to recognize such characters since the automata itself encapsulates the underlying structure of some or all of the curves of a signature.

The five experiments for this problem studied the feasibility of agents as descriptors for signatures utilizing data representing a large number of types of handwritten signature shapes. The first data set comprises 10,000 paths where no part of a path can crossover any other part of the path. The second data set relaxes these criteria and considers 10,000 paths where any part of a path can crossover any other part. The results from both of these datasets were quite promising on randomly generated script.

The question then arose as how to properly locate and align scanned characters with the assumed positions of characters generated by fonts, stored in a digital image database. This line recognition problem utilized a genetic algorithm to determine the number of lines of text in the image and their relative positions to a world coordinate

system. Forty pages of text skewed at different angles test the line recognition genetic algorithm with a high degree of success.

The final problem investigated is the moment-based character recognition. Since OCR systems employ matching algorithms, statistical moment values are typically calculated. The current system computes nineteen intrinsic values, seven Hu and ten Flusser-Suk moments, axis aligned and minimum area rectangle areas for each potential character on a scanned page. The character in the database with the shortest Euclidean distance with respect to the measured values is recognized as the true potential character. The question becomes how to determine the smallest set of measures (or moments) that can still enable character recognition. Here too a genetic algorithm was employed, but the results indicated that this is a difficult problem.

Acknowledgements

I want to dedicate this dissertation to my parents for their love and unwavering support throughout my academic life. Without them, I would have given up my pursuit of a doctoral degree a long time ago. They always believed that I can do anything. Now, I know that it is true. I really wish my father could be here to see me receive my degree but I know he is with me in spirit.

Of course, I could not have completed this dissertation without the help of my friend and colleague Dr. Robert Goldberg. He came along at the right moment. He has been a great mentor and an equally great friend. He nourished my intellect and gave me a push to work a little harder and faster when I needed it most. I hope that I can be half as good a mentor to someone as he was for me.

I want to thank my family for their love, support and encouragement, my sisters Mary and Ann, brothers-in-law Mike and Dave, niece April, Aunt Bernie, Uncle Dave and Cousin Lauren.

I want to thank Dr. Jerry Waxman for his help in my research and teaching. He has always provided help in my Ph.D. when I least expect it and I've thoroughly enjoyed working with him as a colleague for the CSCI 012 course. I also want to thank Dr. Ted Brown for my first teaching opportunity. I now know that I enjoy teaching! Thank you to my examining committee for your questions and suggestions. They helped tremendously in writing my dissertation.

Thanks to my longtime fellow CSCI 012 co-workers Xiaowen Zhang, and Xiu Yi Huang. Xiaowen for hanging out at school and chatting, and Xiu Yi for swapping

teaching hours when I really needed it. To all my friends and colleagues in the Computer Science Department at Queens College, thank you for the chance to relax and talk.

My buddies from Le Moyne College – Mike, Jim, Rene, Dave, Pat, Rob, Tony and Pat, it's time to celebrate after all these years! For my friends at Queens College – Jonathan, Kerry, Ajay, Hema, Nilesh and Arti, thank you for the chance to get away from work and have fun! A special thank you to my friend Jonathan Kramer for allowing me to use without hesitation his school's computing facilities for one of my experiments.

My friend Joe, I've always enjoyed hanging out and watching Star Trek and Babylon 5, Excelsior! To Larry, thank you for the chance to get away from work and chat about anything in your comic book shop. It's always fun to reminisce about Syracuse, N.Y.! Finally, for all of my friends, thank you!

Table of Contents

List of Figures

Chapter 1 Introduction to Genetic Algorithms and Imaging

Many researchers have applied the use of evolutionary computation to solve various image processing tasks (Anderson, Asbury, Gaborski and Tilley 1993; Andre 1994; Guarda, Le Gal and Lux 1998; Liu and Tang 1999; Sural and Das 2001; De Stefano, Della Cioppa and Marcelli 2002; Cha and Tappert 2003). This chapter provides a survey of various aspects of evolutionary computation and the use of evolutionary computation to solve image processing tasks. The survey begins in section 1.1 with a brief discussion of the need for evolutionary computation. Sections 1.2 and 1.3 present a brief description of the two major areas of evolutionary computation, evolutionary programming and genetic algorithms. The first problem addressed in this dissertation uses a genetic algorithm to evolve finite state machines. A finite state machine is a powerful tool that is a program. Programs evolved by evolutionary algorithms tend to have code segments that perform no useful function, called introns. The articles discussed in section 1.4 examine various aspects of introns and the size of evolved programs. Section 1.5 presents an interesting rectangular crossover operator. Section 1.6 presents three articles covering the subject of evolving artificial life organisms. Section 1.7 describes three evolution systems to detect visual features. Section 1.8 contains the description of a mechanism to evolve non-linear image processing filters. Finally, this chapter ends with a discussion of various applications of evolutionary computation to the problem of optical character recognition.

1.1 A Need for Evolutionary Computation

Most computer programs are rigid entities employing deterministic algorithms that do not adapt well to unexpected conditions. Software needs adaptability and the

ability to solve problems that can't be solved using deterministic methods (see figure 1.1). The left side of the table in figure 1.1 lists the attributes of deterministic algorithms while the right side of the table in figure 1.1 lists the attributes of evolutionary algorithms. The field of evolutionary computation developed the paradigm of adaptability, from biological evolution, to have programs evolve solutions, even creation of solutions not planned by the programmer.

Deterministic Method	Stochastic Process
Examines each possible solution one at a time	Examines a set of possible solutions at a time
Rigid, Doesn't easily handle unexpected circumstances	Adaptable
Search guided by mathematical logic	Search guided by random heuristic based on subsequent evaluation

Figure 1.1 - Comparison of deterministic methods and stochastic processes.

The process by which species evolve and create biological organization is known as evolution. DNA is the chemical means to encode the traits and characteristics of a biological organism. Traits pass from organisms in one generation to organisms in the next generation via sexual or asexual reproduction. Variation of current traits and the introduction of new traits within a population occur via mutation and for some species via the recombination from sexual reproduction. Some biological organisms have the flexibility and adaptability to survive in ever changing environments. For the organisms in these resilient species, natural selection is the tendency of the best traits of a species, for survival in the environment, to become more prevalent in the population as a whole.

For individuals, fitness is the ability of that individual to reproduce and pass along its traits to the next generation.

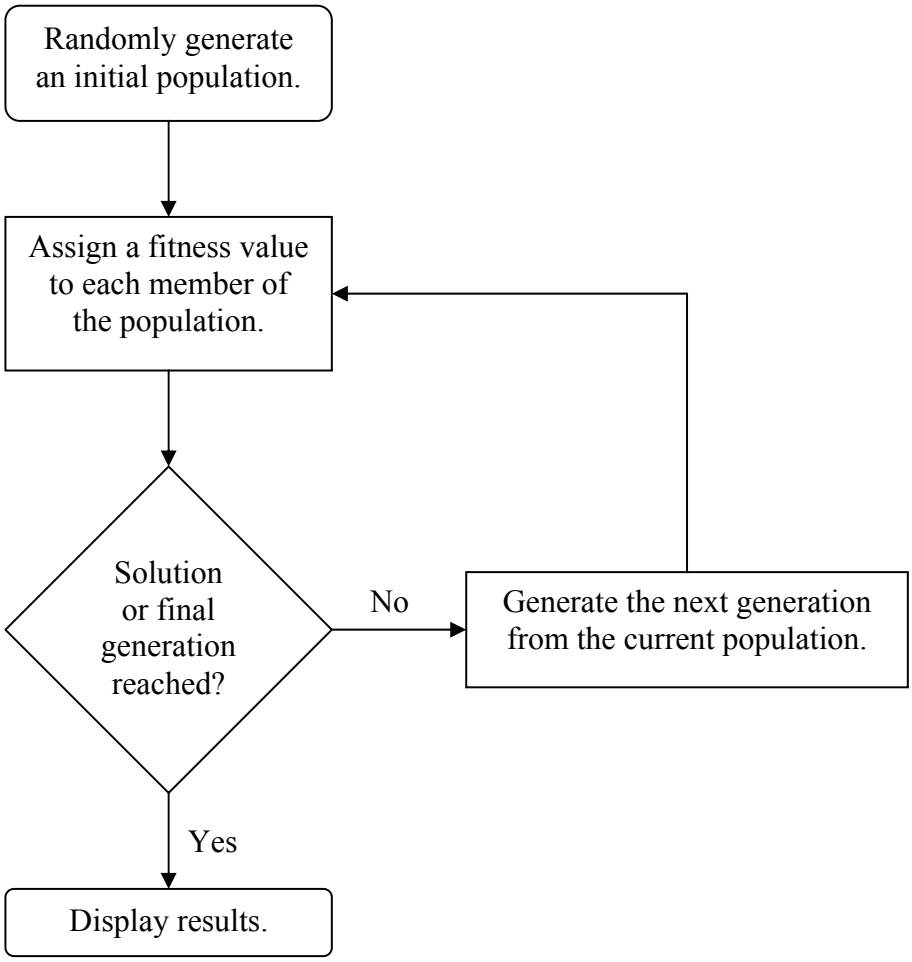


Figure 1.2 - Flowchart of an evolutionary algorithm.

Alex Fraser pioneered the field of evolutionary computation with his series of papers titled *Simulation of Genetic Systems by Automatic Digital Computers* with the first paper (Fraser, 1957) published in 1957. Concurrently, but independently, Hans Bremermann (1958) also encoded the aforementioned mechanisms of biological evolution to use in solving problems in a computer. The flowchart in figure 1.2 illustrates the general structure of an evolutionary computation algorithm. Several years

after Fraser's initial work and Bremermann's initial work in evolutionary computation, the area of evolutionary programming developed as a part of the field of evolutionary computation. The next section provides a brief description of evolutionary programming.

1.2 Evolutionary Programming

One of the founding fathers of the field of evolutionary computation and a major contributor to the subsequent development of the field was Lawrence Fogel (1928-2007). Fogel's initial interest was the characterization of intelligent behavior, a theme he presented in his doctoral dissertation at UCLA and concentrated on in three editions of his landmark text *Artificial Intelligence through Simulated Evolutions*. Fogel proposed evolutionary algorithms to predict one's environment and enable experimental observations of simulated behaviors. Fogel suggested finite state machines as the means to process the environment inputted as a sequence of user-defined alphabet symbols. Then, evolutionary algorithms applied to finite state machines evolved the resulting intelligent responses or strategies (Fogel, Fogel and Atmar, 1991).

A finite state machine consists of *n* states and a finite set of input and output symbols. For each state, each input symbol maps to an output symbol and a transition to another state (see figure 1.3). The finite state machine begins in an initial state and receives a sequence of input symbols. For each input symbol fed into the finite state machine, the finite state machine generates an output symbol and makes a transition from the current state into the next state. For example, if the current state is 0 and the input symbol is 1 then the output symbol is "A" and the next state is 2. The finite state machine assigns the current state the next state value of 2 and the finite state machine

awaits the next input value. Implementation of an finite state machine in a computer program is a two-dimensional array with the number of rows equal to the number of states and the number of columns equal to the number of input symbols, see figure 1.4. Each element of the array consists of an output symbol and a next state value.

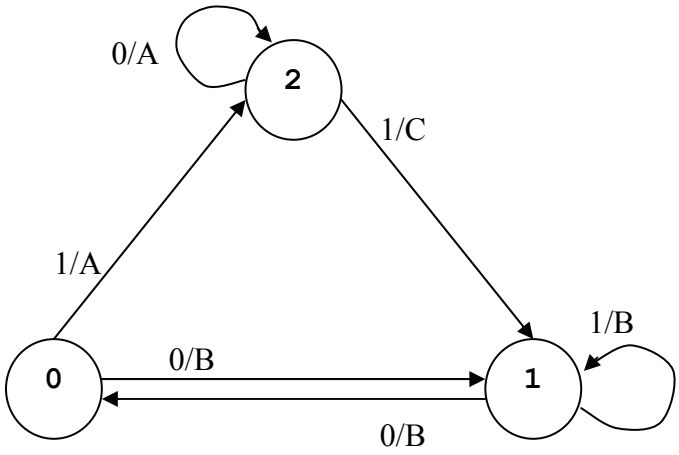


Figure 1.3 - A finite state machine.

		Inputs	
		0	1
States	0	B/1	A/2
	1	B/0	B/1
	2	A/2	C/1

Figure 1.4 - Finite state machine transition table.

The input and output symbols of a finite state machine aren't limited to simple characters and can map to any kind of object. Finite state machines are powerful tools and computationally complete because they can represent any program or algorithm, for an algorithm, assuming there is a halting condition. The complexity and variety of finite state machines is amazing. The number of possible finite state machines with 3 states, 2 input symbols and 3 output symbols is approximately a half million. There is a dramatic

increase in the number of finite state machines by adding just 2 more states and 1 more input symbol. The number of possible finite state machines with 5 states, 3 input symbols and 3 output symbols is approximately 438,000,000,000,000,000. Wirt Atmar described all these statistics about finite state machines in his 1976 Ph.D. dissertation titled *Speculation on the Evolution of Intelligence and Its Possible Realization in Machine Form*.

Feeding a finite state machine a sequence of input symbols and measuring the finite state machine's performance generates the fitness value of the finite state machine. A selection method uses the fitness values to choose a parent for reproduction. Applying some form of mutation to the parent creates an offspring. Mutation can take the form of changing an output symbol, altering a state transition, adding a new state, or deleting an existing state.

Although one applies evolutionary programming to evolve the strategies embedded in the finite state machine, Fogel, Angeline and Fogel (1994), based on Fogel (1993) and Bäck, Rudolph and Schwefel (1993), aptly points out two inherent differences between evolutionary strategies and evolutionary programming. Evolutionary strategies encapsulate coding structures as individual members of a population and employing "reproductive" operators to generate new strategy. However, evolutionary programming encodes abstract structures as individual species. A second important respective difference is the deterministic versus stochastic stress of the selection process. Evolutionary strategies rely on a deterministic selection based on the fitness of the individual, whereas evolutionary programming conducts a random tournament amongst the possibilities for determination of the species survival in future generations. A

description of the latter is in Fogel, Fogel, Atmar and Fogel (1992). In addition, Koza (1993) argued that main task artificial intelligence machine learning is the automatic and dynamic decomposition of larger problems into smaller ones in order to simplify the solution process.

Finally, the usage of simulated evolution as a predictor, and in particular forecasting, was a revolutionary concept because one utilized evolution in its classic form as a means of describing history to explain what happened and how the current situation arrived. The contribution by Lawrence Fogel is the first documented time that one utilized evolution as a mechanism to predict future events (predicting the existence of signal in noise). The rest of this section presents a sample of Lawrence Fogel's work.

Fogel, Fogel and Porto (1990) applied evolutionary programming to the problem of training neural networks on each of two problems, the XOR problem and the gasoline blending problem. Neural networks are parallel processing nodes interconnected with fixed or variable weights that are typically trained with a back propagation algorithm. In this work, the individuals in the evolutionary population are vectors of the neural network weights rather than the usual finite state machines. The author's evolutionary program converged to a solution in 1/5 the time of the typical back propagation training algorithm. Also, Porto, Fogel and Fogel (1995) employed evolutionary programming, simulated annealing, and the back propagation algorithm on sonar signals in order to discriminate between man-made objects and natural underwater objects. Both evolutionary programming and simulated annealing surpassed the results of the back propagation training method.

Tuning various parts of an evolutionary program system can itself be done using an evolutionary program. Fogel, Fogel, and Atmar (1991) investigated the use of a meta-evolutionary program to tune the parameters of a concurrently running evolutionary program working on a particular problem. Another example of evolutionary program tuning is self-adaptation of finite state machines (Fogel, Angeline and Fogel, 1995 & 1996) within an evolutionary programming system. Self-adaptation of the finite state machines adapts the use of the mutation operator in an evolutionary programming system. Recall that there are five mutation methods available in the system: add a state, delete a state, change the start state, change an output symbol, and change a next state transition. The mutation parameters of an evolutionary programming system determine the mutations made to a parent in order to create an offspring. The mutation parameters are: 1) the number of mutations, 2) for each mutation, the mutation method and 3) choice of the finite state machine component to mutate. Self-adaptation evolves the mutation parameters in the same way that an evolutionary program evolves finite state machines.

Evolutionary programming has its military applications. An evolutionary program (Fogel, Fogel, and Atmar, 1993) optimizes a firing sequence for a collection of anti-satellite weapons to disable or destroy the largest number of enemy satellites. The fitness function computes measures for a certain firing sequence based on the weighted sum of the time of destruction and the importance of each satellite.

An evolutionary program (Porto, Fogel and Fogel, 1998) can also optimize tactics for platoon led tank maneuvers. The parameters to optimize are the probability of killing the enemy, the probability of getting killed, the probability of arriving at an objective on

time, and minimizing the distance to an objective. The evolutionary program also optimizes the priority of each parameter.

Next, Porto, Fogel and Fogel (2004) applied evolutionary programming to the problem of constructing and training neural networks to classify sensor signals received by a set of remote sensors distributed on the battlefield. The results of the experiments in this paper show promise in the use of an evolutionary program to the training of neutral networks for seismic signal detection.

Finally, Porto, Fogel, Fogel, Fogel, Johnson and Cheung (2005) applied evolutionary programming to the problem of constructing and training neural networks as well as rule based sets to classify sonar signals to detect sea mines. The results of the experiments in this paper show promise in the use of an evolutionary program to the training of neutral networks as well as rule based sets for sonar detection of sea mines.

Evolutionary programming also has its civilian applications. A set of rules control the rate at which cars enter a freeway at various entrances. The rules use information about the number of cars across the entire length of the freeway. McDonnell, Fogel, Fogel, Rindt and Recker (1995) use an evolutionary program to optimize the thresholds used by the control rule set. The evolved rule set performed as well as the standard rule sets created for freeway control systems.

McDonnell, Page, Fogel and Fogel (1997) developed an evolutionary program to schedule the delivery of fuel from a central terminal to a number of local stations. The values to optimize are each station's delivery window, the use of company or non-company delivery trucks, and whether the shift is during the day or night. The authors

found significant cost savings with the evolutionary program generated schedules as compared to human generated schedules.

Finally, the last two problems in this section involve signal processing and computer process scheduling. Fogel and Fogel (1996b) developed an evolutionary program to determine if random signals exist within linear and non-linear noisy data. The evolutionary program optimizes the set of parameters that best fits the time series with the experiments indicating promising results. In the same year, Fogel and Fogel (1996a) developed an evolutionary program to schedule jobs on a set of computers with diverse architectures and run-time performance. A load balancing algorithm may not effectively distribute jobs across a heterogeneous set of computers. Two different greedy algorithms perform better than the load balancing algorithm. But, the evolutionary program performed better than the greedy algorithms 90% of the time. The authors also tested the evolutionary program to see if it can perform better when initialized with the best schedules of the best greedy algorithm.

After reviewing the many papers by Lawrence Fogel and read about his many contributions to the field of evolutionary computation, one paper personally stands out for me in that one should consider it a timeless classic on evolutionary computation (Fogel, Fogel, and Atmar, 1991). The reason I chose this particular paper is that it provided a serious research protocol on experimental studies in this field and the analysis provided has far-reaching conjectures and conclusions that all future evolutionary computation studies should consider.

Lawrence Fogel's main work bred finite state machines to solve various problems including signal processing and entertained the idea of character recognition. This thesis

can be thought of as falling under this context with the main contribution of this thesis strongly complimenting Fogel's earlier work but this thesis also investigates considerations beyond his original contributions. Whereas Fogel bred finite state machines which were transducers that provided output at every processing step of the input, the finite state machines in chapter 2 of this thesis are recognizers that make a final decision after processing all input. Also, his research was to understand the extent of evolutionary algorithms providing solutions to important problems. The main contribution of this thesis is the extensive experimentation on a number of variations of the underlining problems in optical character recognition. As such, this thesis devised an experimental approach to the research as opposed to a more mathematical approach. The next section provides a brief description of the other area of evolutionary computation known as genetic algorithms.

1.3 Genetic Algorithms

John Holland (1975) described in his book *Adaptation in Natural and Artificial Systems* an implementation of an evolutionary algorithm that differs in certain ways from an evolutionary program. Genetic algorithm is the name of Holland's evolutionary algorithm. Recall that an evolutionary program evolves finite state machines, programs, such that the execution of a finite state machine provides a potential solution to the problem. On the other hand, a genetic algorithm evolves a population of potential solutions directly. A bit string called a chromosome, the equivalent of its biological counterpart (see figure 1.5), represents each individual solution in the genetic algorithm population. The problem's fitness function receives, as input, the chromosome of each

individual of the population generating a corresponding output value to use as the individual's fitness value. The fitness function measures how closely the individual represents the actual solution to the problem. Once the fitness evaluation completes for the entire population, the genetic algorithm creates the next generation of offspring. Creation of an offspring requires two parents with the parents randomly chosen from the current population via a particular selection method. Genetic algorithms introduce a new reproduction genetic operator known as crossover. Crossover randomly chooses one or more points within a chromosome to allow sections of the chromosome to swap between both parents to create new offspring. A genetic algorithm also has a mutation genetic operator which operates on the bits of the chromosome, flipping bits according to the genetic algorithm's mutation probabilities.

```
01101000010111010111100100101000
```

Figure 1.5 - Chromosome, the genetic algorithm solution model.

Originally, genetic algorithms evolved solutions directly. A new field of evolutionary computation, genetic programming, sprung from the adaptation of a genetic algorithm to evolve programs. Since finite state machines are programs, it is reasonable to examine a few applications of a genetic algorithm evolving finite state machines. In addition, the first problem investigated in this dissertation also uses a genetic algorithm to evolve finite state machines to recognize handwritten signatures.

The goal of Ladd's (1995) robotic ant is following a trail of food for a certain amount of time. A two-dimensional grid of squares defines the ant's environment with the edges of the grid forming cliffs that the ant can not cross. The top of the grid is north and each square is empty or contains food. The ant begins its journey in the center square

on the top row and faces south, one of four possible directions. The "view" in front of the ant determines the ant's next move. The views are a food square, an empty square, or a cliff and the moves are step one square ahead, turn left 90°, or turn right 90°. If the square ahead contains food, the ant eats the food, adding one to its fitness value, and the square changes to empty. A finite state machine defines the actions of the robot ant and is its best strategy for finding food. The finite state machine for an ant has an input alphabet of three view symbols (F for food, E for empty, C for cliff) and an output alphabet of 3 move symbols (A for ahead, L for turn left, R for turn right).

A genetic algorithm creates robotic ant finite state machines via selection initialized with a randomly generated population. Sixteen parameters control the genetic algorithm. The master grid encoded as a two-dimensional array where an element containing "1" represents a food square and an element containing "0" represents an empty square. Calculation of the fitness value for an ant finite state machine uses a copy of the master grid. The ant finite state machine executes moves until the ant reaches the bottom of the grid or executes the maximum number of moves. Linear normalized fitness scaling adjusts the fitness values of the ant finite state machines. Elitist selection copies the best ant finite state machine to the next generation with the use of mutation to create the rest of the next generation of ant finite state machines.

The best ant finite state machine created by Ladd's (1995) genetic algorithm scored 35 food points out of a possible 40. An examination of the movements made by this ant finite state machine demonstrates the evolution of an eastward and westward backtrack algorithm that regains contact with the food trail by moving towards the cliff in a diagonal direction without going over its previous path. A further inspection of the ant

finite state machine's movements reveals a looping mechanism the ant finite state machine uses to travel to the east. These intricate actions illustrate that the intelligence of the ant finite state machine evolved by natural selection rather than creation by random chance. Several runs of the genetic algorithm show similar types of behaviors evolving in the most successful ant finite state machines.

Aporntewan and Chongstitvatana (2000) developed a hardware based genetic algorithm to evolve finite state machines that mimic sequential circuits. One specially designed CPU executes all the tasks of the genetic algorithm except for the fitness function which executes on up to 8 co-processors. Training of the finite state machine compares the output sequence generated by the finite state machine with the output sequence generated by the actual circuit, for each input sequence used by the fitness function.

Hikage, Hemmi and Shimohara (1997) developed a progressive evolution system to have individuals acquire the successful skills to solve their problem in several steps. The authors used the artificial ant in the Muir trail problem to illustrate their concept. The ant examines the five squares in front and on either side of it. The squares are either empty or contain a food tablet. The ant has 3 possible moves. There are 96 combinations of the contents of the 5 squares and the 3 possible moves. The authors refer to these combinations as action primitives. The evolutionary algorithm attempts to evolve an ant that can traverse the entire Muir trail in the proper order. The final evolutionary step is the grid containing the Muir trail. The intermediate steps are the grid containing other trail(s) designed to train the ant to acquire some of the action primitives that are

necessary for success during the final step. The number of action primitives needed to learn an intermediate step is the measure of the evolutionary complexity of that step.

Langdon (1998) implemented the artificial ant and Santa Fe Trail problem with a modification that coerces the ants to traverse the trail in the correct order. The modification does not place the entire trail in the grid but adds succeeding food pellets as the ant consumes the preceding food pellets.

Programs evolved by a genetic algorithm or genetic programming tend to have code segments, called introns, that do not contribute to the program's fitness value. The next section examines the effect of introns on evolved programs.

1.4 Introns and Program Size

Introns (see figure 1.6) exist in variable length genetic programming as program statements that do not affect the program's outcome. The counterparts of introns are exons, program statements that do affect a program's outcome. The next subsection examines the ability to measure the compression of information in a genetic program, its relationship to introns, and the need to balance the amount of compression in order to evolve short well-designed programs. Subsection 1.4.2 addresses the issue of managing exponential code growth in relation to introns.

```
x = 1 * x
```

Figure 1.6 - An intron example in genetic programming.

1.4.1 Introns and Compression Pressure in a Typical Genetic Program

Nordin, Banzhaf and Francone (1997) examine program size from the viewpoint of the usefulness of introns. Learning utilizes compression of information. For example, a theory formulated from a collection of data yielded from an experiment. Evolutionary computation is a type of learning with individuals adapting to a better general representation of a data set. Nordin, Banzhaf and Francone (1997) assert that compression pressure exists within variable length genetic programming favoring compact simpler solutions. The intrinsic compression pressure within a genetic programming system controls the amount of compression within a program. Genetic programming attributes such as a program's representation, the genetic operators, and the probability parameters, control the compression pressure's strength. The amount of compression pressure shapes a program's generality and adaptability. Shorter programs tend to represent a more generic solution that is valid for data not even seen by the program. The ability to produce short generic solutions is a positive outcome of compression pressure. But, a compression pressure strength that is too high can have a negative impact on the genetic programming system's convergence. Hence, a strong compression pressure leads to much shorter incomplete solutions rather than longer complete solutions. Being able to measure compression pressure can help to balance its negative and positive aspects.

Since introns exist in genetic programs, the absolute length of a program is not a good measure of the compression pressure. A program's effective length is the program's absolute length minus the length of all the introns in the program. The

effective length is essentially the length of all the exon program code. Effective length is not the only value measuring compression pressure. The change in program fitness from generation to generation also measures compression pressure.

Recall that the individuals with higher fitness values have a higher probability of selection to generate offspring. Crossover swaps contiguous code segments between two parents to produce two offspring. The change in fitness value between parents and its offspring occurs in one of three directions: negative, zero or positive. If the two code segments swapped are introns or identical exons, the offspring's fitness value remains unchanged. If the code segment swapped out is more useful than the code segment swapped in, the offspring's fitness value is worse then its parent. Finally, if the code segment swapped out is less useful than the code segment swapped in, the offspring's fitness value is better then its parent.

The effective fitness of a program is the number of the program's offspring chosen for reproduction in the succeeding generation. Increasing a program's effective fitness decreases the area of useful code segments damaged by crossover. A decrease in effective length, an increase in absolute length, or both can increase a program's effective fitness, increasing the chances that its offspring inherits intact useful code segments from its parents. The addition of introns into a genetic program increases the program's absolute length. Nordin, Banzhaf and Francone (1997) contend that introns are a program's attempt to defend against the destructive affects of crossover. Nordin, Banzhaf and Francone (1997) tested this theory on three data set problems. The genetic programming system performed 160 runs for each problem data set with a population size of 3,000 individuals evolving for 40 generations. The average fitness value

decreases, average absolute length of an individual increases exponentially in the later generations, and effective length decreases early and levels off towards the end of the run. The addition of introns adds to a genetic program's size but it helps to carry over the effective portions of the genetic programs into the next generation. But, a program's size may affect the program's generality, the ability of the program to compute an accurate solution for unanticipated types of input data. Introns can be a useful part of a genetic program. But, the resulting code growth is a problem as addressed in the next subsection.

1.4.2 Introns and Code Growth

Exponential code growth, called code bloat, in genetic programming is a problem because it affects the amount of time to compute the fitness of individuals and reproduce offspring with little improvement in fitness. Elimination or a vast reduction of introns from programs can help the genetic programming search process.

Smith and Harries (1998) use explicitly defined introns to study the "replication accuracy force" (RAF) stated by both Soule and Foster (1997) and McPhee and Miller (1995). RAF works in genetic programming to introduce intron code into a program as a defense against the destructive effects of crossover. Then, Smith and Harries (1998) developed an improved fitness selection method to eliminate introns from programs only to discover exon program code with intron-like behavior. Finally, to further curb code growth, the authors made changes to the improved fitness selection method as well as creating new crossover operators that do not favor large fitter programs.

Iba and Terao (2000) use genetic programming to evolve cooperative behavior between multiple agents to solve various problems such as robot navigation. The size of

the evolved programs is important. Larger programs take longer to run and need more space in memory. Iba and Terao (2000) propose a method of removing "effective" introns, segments of non-executable code, via identifying the symbols in the program tree that never execute. Attach an execution count to each symbol in the tree. After program execution remove the sub-trees anchored by the symbols with a count of zero. The authors found removal of effective introns made a positive difference in the increase of fitness values, limiting code growth and producing solid performing programs.

Boryczka (2005) used ant colony programming to solve approximation problems in each of two approaches: expression and instructional. An ant colony system uses a collection of artificial ants that work together to find the solution to a problem. Each ant has a limited amount of memory, makes each move according to a probability function, and works in discrete time steps. Ant colony programming is a combination of an ant colony system and genetic programming. In the expressional approach, the approximation function is a set of tree-based expressions in prefix notation. In the instructional approach, the approximation function is a set of single operator arithmetic assignment statements in sequential order. The elimination of introns is a post-processing step applied to each approximation function in the new generation to recognize redundant or insignificant expressions (introns) within the approximation function and replace the introns with the proper value of 0 or 1, or a simplified version of the expression. Boryczka's (2005) experiments revealed that, in the expressional approach, the final approximation function ran 25% faster with intron elimination than without intron elimination and, in the instructional approach, the final approximation function ran 75% faster with intron elimination than without intron elimination.

While the previous three researchers worked to remove introns to eliminate code growth, Luke (2000 & 2003) states that the two theories of code growth caused by introns, "defense against crossover" and "removal bias", focus mainly on inviable code introns. Luke's theory states that introns are not the sole cause of code growth. Luke identifies two types of introns. The first type of intron is inviable code whose removal changes the program's operation due to the fact that another code segment in a different place in the program nullifies the inviable code's effect on the program. The second type of intron is un-optimized code whose simplification doesn't change the program's operation. Both theories, "defense against crossover" and "removal bias", state that crossover takes place inside inviable code to protect against destroying viable code segments in the program. Luke's experiments restrict crossover to non inviable code segments of the programs. The results of the experiments show that code growth continues unabated. Crossover points lower in the program tree have a smaller affect on the program's fitness value than crossover points higher up in the tree. A secondary reason for code growth is the depth of the chosen crossover point. Larger trees provide more crossover points to limit the damage to a program's fitness value from crossover.

Not all genetic programming systems have significant code growth. Miller (2001) determines why significant code growth does not exist in a new form of genetic programming, Cartesian Genetic Programming, CGP. The code in a CGP program takes the form of a rectangular array of nodes. Each node can implement any form of programming structure which operates on its inputs. Miller (2001) tested two types of CGP programs. In the first type of CGP program, the typical CGP program, the nodes form a partially connected graph with redundant nodes. To balance the test, in the second

type of CGP program, the nodes form a fully connected graph without redundant nodes. The experiments found that the redundant nodes provided a type of compression pressure which helped limit code growth and improve program fitness. In addition, the typical mutation operator provides a type of parsimony pressure.

The individuals of the genetic algorithm in the first problem of this dissertation are rectangular in format. The genetic algorithm uses a rectangular crossover operator on the individuals in that first problem. The next section contains another example of a rectangular crossover operator.

1.5 A Rectangular Crossover Operator

Wallet, Marchette and Solka (1996) developed a crossover operator for a genetic algorithm in which a matrix stores an individual's two-dimensional chromosome, closer to the individual's "true" structure. Wallet, Marchette and Solka's (1996) crossover operator randomly selects a rectangular subsection of each parent's chromosome and swaps the area within the subsections between the two parents to form two offspring, see figure 1.7. For individuals stored in a linear format, the disruption rate for one point crossover is not rotational invariant. Assume the individual consists of R rows, C columns, and N=R*C+1. For any interior location of the individual, the disruption rate for the neighbors above or below the location is 1/R and for the neighbors to the left or right of the location is C/N. For individuals stored in the matrix format, the disruption rate for Wallet, Marchette and Solka's (1996) crossover operator is close to rotational invariance especially when R=C leading to improved performance for the genetic algorithm. For any interior location of the individual, the disruption rate for the

neighbors above or below the location is 1/((R-1)*R) and for the neighbors to the left or right of the location is 1/((C-1)*C).

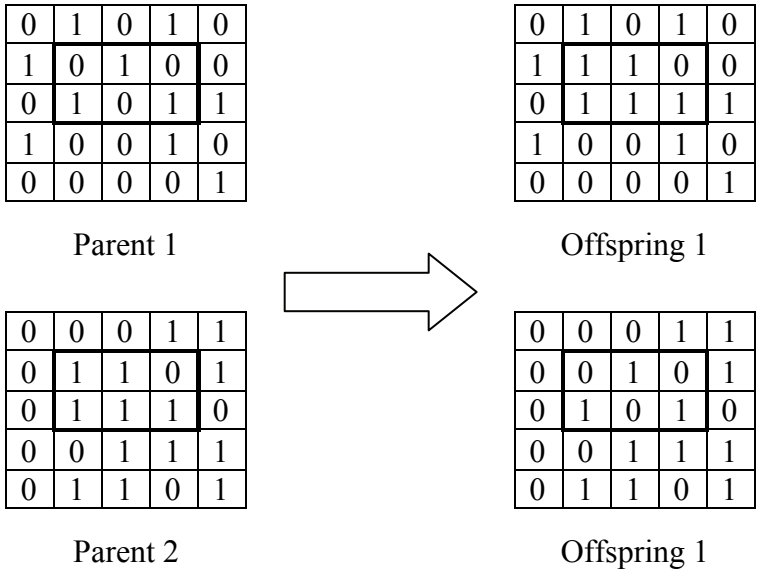


Figure 1.7 - An example of matrix crossover.

Reward	Conditions
4 points	The location's value matches the value of the corresponding location in the key matrix. The values of all four neighbors match the values of the corresponding locations in the key matrix.
1 point	The location's value does not match the value of the corresponding location in the key matrix.
None	The location's value matches the value of the corresponding location in the key matrix. The value of at least one neighbor does not match the value of its corresponding location in the key matrix.

Figure 1.8 - Abstract program's reward scheme.

In the first set of experiments, the genetic algorithm (GA) worked on an abstract problem that uses a randomly generated binary key matrix. The GA searches for the best binary matrix individual that closely matches the key matrix. An individual's fitness

value is the sum of the rewards awarded to each location in the individual. Compute a location's reward from the location's value and the value of its four neighbors, see figure 1.8. Figure 1.9 lists the parameters and values for the two experiments of the GA. The matrix size of each individual in experiment 1 is 10×10 and each genetic algorithm run executes for 100 generations. The matrix size of each individual in experiment 2 is 20×20 and each genetic algorithm run executes for 500 generations. Each experiment works on a population stored in a linear format and then works on a population stored in a matrix format. Plot the average of the most-fit individuals from each of the 100 generations for both formats. In both experiments, the matrix format performs significantly better than the linear format. The matrix format produces individuals with greater fitness values than the linear format individuals.

Parameters	Value
Runs	100
Population size	100
Crossover rate	0.6
Mutation rate	1/N
Selection	Fitness proportional
Elitist selection	Yes

Figure 1.9 - Abstract program's genetic algorithm parameter values.

$$\sum_{m,n} \lambda abs(\hat{x}_{m,n} - x'_{m,n}) + (1-\lambda) abs(\hat{x}_{m,n} - \hat{\bar{x}}^{S-}_{m,n})$$

Figure 1.10 - Image de-noising genetic algorithm fitness function.

In the second set of experiments, the genetic algorithm (GA) worked on an optimization problem of removing noise from an image of size m×n where each pixel is an integer value in the range 0 to 255 representing 256 gray levels. Figure 1.10 contains

the genetic algorithm's fitness function that measures the spatial dependency in the estimated image and the similarity between the estimated image and the original image. The variable x' represents the original noisy image, $\hat{x}$ denotes the evaluated image, $\bar{x}^{s-}$ is the average of the four neighbors of a particular pixel, and λ is the smoothing parameter. Figure 1.11 lists the parameters and values for the de-noising experiment of the GA. The smoothing parameter λ is a value between 0 and 1. Choosing the right value for λ is a balance between removing noise and over-smoothing. The optimal image should be the original image when λ is 1 and the optimal image should be an even gray level image when λ is 0. The mutation operator increases a pixel's value slightly if the pixel's value is lower than the average of the pixel's eight neighbors. The mutation operator decreases a pixel's value slightly if the pixel's value is higher than the average of the pixel's eight neighbors. The algorithm's initial population is a set of images slightly altered from the original image. The best image at the end of the evolution had considerably less noise than the images in the initial population while showing consideration for the edges and object's features in the image. The remaining sections in this chapter provide examples of evolutionary computation applied to various problems in image processing starting in the next section with the evolutionary computation area of artificial life.

Parameters	Value
Image size	64×64
Population size	50
Generations	150
Mutation rate	Slightly higher than 1/N.
λ	0.2

Figure 1.11 - De-noising program's genetic algorithm parameter values.

1.6 Artificial Life

The discussion in this section involves the use of a genetic algorithm to evolve a population of artificial life organisms to solve various problems. The first subsection, 1.6.1, discusses the application of artificial life to binary image correction. The second subsection, 1.6.2, examines general genetic algorithm applications to artificial life problems. The final subsection, 1.6.3, reviews a special type of artificial life organism known as a self-replicating cellular automaton.

1.6.1 Binary Image Correction

Thearling (1992) worked to improve the fitness of a group of artificial life organisms that correct errors in a bit-mapped image by having the organisms communicate between each other. The image is a two-dimensional array of pixels whose value is "on" or "off" and each pixel has an artificial life organism. The errors in the image are one of two possible types, unidirectional or bi-directional. Unidirectional errors are pixels whose value is "off" when it should be "on". For bi-directional errors, the value of a pixel should be "on" when it actually is "off" or vice versa.

Each organism has a home pixel, a unique pixel in the image. Each organism travels through the neighboring pixels to determine whether or not to correct its home pixel. A finite state machine defines each organism's chromosome. The following items: the original value of the home pixel, the current correction decision for the home pixel, and the original value at the current pixel location form the finite state machine's input symbols. The finite state machine's current state combined with the current input

symbol results in the finite state machine moving to the next state and producing an output symbol. The following items: Forward/Backward movement of the organism, Left/Right movement of the organism, and the new correction decision for the home pixel form the finite state machine's output symbols. When the organism's finite state machine enters its final state the organism's last correction decision is its final decision with a lifetime limit to prevent organisms whose finite state machine may never reach the final state.

The genetic algorithm works with a population of randomly generated organisms where each organism works on a slightly different corrupted image. Each pixel in the organism's image has its own copy of the organism and the organism's copies form a sub-population. When all organisms in a sub-population reach their final state or lifetime limit, the genetic algorithm calculates the sub-population's fitness by totaling the number of correct pixels in the image. With the fitness scores for the population sorted, reproduction of the next generation uses the best X% of the population. The percentage of offspring an organism generates is equal to the organism's fitness value divided by the sum total of the fitness values of the best X% of the population. An organism chooses one mate to create each offspring using two-point crossover and one point mutation.

Images with unidirectional errors show a high rate of error detection and correction because pixels that are "on" are always correct and the organisms use this as a frame of reference to make a good determination to correct its home pixel. But, as the error rate in the image increases then the error detection rate decreases because it is harder for the organisms to orient themselves using correct pixels. Organisms in an image with bi-directional errors can not rely on "on" pixels being always correct.

Therefore, the error detection rate for such images is low. An organism can easily determine if its home pixel is correct when it is in an area containing a large number of correct pixels. This determination is a form of communication between organisms. Adding information to the finite state machine's input symbols incorporates this communication into an organism's finite state machine. The following items now form the finite state machine's input symbols: the current state, the original value of the home pixel, the current correction decision for the home pixel, the current location's original value or if the location's organism is done then the current location's corrected value, and the completion status of the current location's organism. The addition of communication only improved, by 5%, the fitness of a sub-population working on an image with bi-directional errors. The next subsection contains a study of artificial life in the framework of genetic algorithms.

1.6.2 Genetic Algorithms and Artificial Life

The scope of artificial life is large enough to warrant an examination of the non-image processing problem of this subsection along with the examination of the two image processing problems in the previous and next subsections of this section. Mitchell and Forrest (1994) examined genetic algorithms as models of natural phenomena and provided examples of overlap between genetic algorithms and artificial life. The first example involves learning and the Baldwin Effect. Learning is an adaptive process that spans an individual's lifetime whereas evolution is an adaptive process that spans the development of life on earth. How do these two processes interact and can learning influence evolution? The Baldwin effect states that learning does not directly encode into

one's genes but learning does increase the odds that an individual will survive long enough to create offspring who've received genes from their parents that are responsible for learning. An artificial life system that models the interaction of learning and evolution can use a neural net to model learning and a genetic algorithm to model evolution by using a population of neural nets.

The next two examples Mitchell and Forrest (1994) examined involve modeling the behavior of ecosystems and studying the evolutionary dynamics of populations. Echo is an abstraction of a real ecosystem to form a simple ecosystem comprising a lattice of sites populated by agents and renewable resources. Agents interact between each other via trading, fighting, and mating. Each type of agent consumes and stores different types of resources. Trading and fighting form a kind of exchange of resources between two agents. Mating generates offspring between two agents. An agent can replicate itself. The agent's chromosome defines rules of interaction with other agents plus the types of resources it can consume. External appearance enables agents to develop social rules and mimicry. A study of the Echo system provides an idea of the flow of resources in a system and the cooperation and competition between groups of agents. Strategic Bugs is a two-dimensional world of adaptive agents and renewable food. The agents eat food, store food in a reservoir as energy and use the energy for movement or reproduction. The agent dies when its energy reservoir is empty. The design of the Bug's chromosome keeps track of the usage of each gene. Strategic Bugs measures a population's evolutionary activity by examining the rate at which the population incorporates useful genetic innovations and tracking a population's consistent use of new genes. The next

subsection deals with the third article in the survey of work into artificial life organisms and genetic algorithms.

1.6.3 Self-Replicating Cellular Automata

Lohn and Reggia (1995) use genetic algorithms to discover various self-replicating structures. Normally, cellular automata (CA) model self-replicating structures but Lohn and Reggia (1995) use an effector automata (EA), a modified form of the CA model. A grid of cells forms cellular space with each cell occupied or empty. In a CA, an automaton inhabits a cell whereas, in an EA, an automaton is the cell. The automata work in parallel, accept input values from the surrounding cells, and execute a rule to generate an output value. In a CA, an automaton outputs an internal state transition whereas, in an EA, an automaton outputs an action to execute. The cells to the top, right, bottom, and left form a cell's neighborhood. Letters represent the different types of automata with each type having the same set of rules. The set {A, B, C, D} describes an EA with four different automata types where the multiplicity of "a", M_a^t, defines the number of automatons of type "a" in a simulation at time t.

A rule table (see figure 1.12) stores the set of condition-action rules describing an automaton's behavior. The format of a condition-action rule is CTRBL $\rightarrow$ *action*, where CTRBL denotes the automata located in the Center, Top, Right, Bottom, and Left cells. Figure 1.13 lists the four possible actions an EA executes. CBA•C $\rightarrow$ NULL is an example table entry whose action NULL executes when a cell's automaton is type C, the top neighbor cell is type B, the right neighbor cell is type A, the bottom neighbor cell is empty, and the left neighbor cell is type C. If two automata try to occupy the same cell,

mutual annihilation destroys both automata leaving the cell empty or a random winner policy chooses which automaton occupies the cell.

CTRBL

A●●●●	action	rules for automata-type **A**
AA●●●	action	
...	...	
ACCCC	action	
B●●●●	action	rules for automata-type **B**
BA●●●	action	
...	...	
BCCCC	action	
C●●●●	action	rules for automata-type **C**
CA●●●	action	
...	...	
CCCCC	action	

Figure 1.12 - Rule table chromosome in the genetic algorithm, Lohn and Reggia (1995) page 2.

Action	**Description**
MOVE <dir> <rot>	Move one cell in the specified direction and rotate the specified number of degrees
DIVIDE <dir> <rot> <dir> <rot>	Divide into two daughter automata according to the specified directions and rotations
DESTRUCT	Cease to exist
NULL	No action

Figure 1.13 - Actions used in current EA model, Lohn and Reggia (1995) page 3.

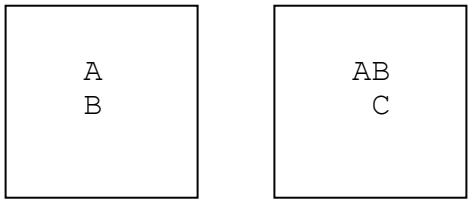


Figure 1.14 - Two examples of seed structures; a seed structure is a group of automata placed in the grid of cells to begin the self-replication process.

The rule table forms the chromosome for individuals in the genetic algorithm population. The crossover point for single point crossover divides the rule table at a specific rule and mutation changes the action for a randomly chosen rule in the table. Seed structures (see figure 1.14) test the chromosomes by running a simulation for 10 time steps and calculating the chromosome's fitness value. The fitness function $F = w_1 f_{gm} + w_2 f_{rpm}$ is a weighted combination of two fitness values, f_{gm} is a growth measure and f_{rpm} is a relative-position measure, where $w_1 + w_2 = 1$, $w_1 \geq 0$, $w_2 \geq 0$. The growth measure function f_{gm} gauges the creation of individual automatons during the simulation. The relative-position function f_{rpm} gauges how the adjacencies between individual automata in the current structure are similar to those in the seed structure. Lohn and Reggia (1995) executed the genetic algorithm for 75 runs of 2000 generations each and discovered six different self-replicating structures consisting of 2, 3 and 4 different automata. The next section surveys three systems to evolve mechanisms to detect visual features.

1.7 Evolving and Detecting Visual Features

A computer vision system can detect boundaries, one type of feature. It is possible that a vision system needs to determine the distance to an object at the center of the system's field of vision, as seen in subsection 1.7.1. The construction of face tracking feature detectors is a more complex task then simple determination of a distance to an object. Subsection 1.7.2 discusses a complete feature tracking system. Subsection 1.7.3 closes this section with an investigation of autonomous agents working in parallel to detect features in an image.

1.7.1 Detecting an Object's Distance

An interest operator uses simple correlation methods to expose key points and compute precise distance information for objects in an image. A mobile robot could use an interest operator to help adapt its vision system to map and extract vital information from its dynamic environment. Specifically, Ebner (1998) uses a genetic programming system to create an interest operator that closely matches a Moravec interest operator. The Moravec interest operator utilizes the minimum of the sum of the squared differences of the eight adjacent pixels to determine if the center pixel is a local maximum representing a possible key point in the image. Subsection 1.7.1.1 describes the aspects of the genetic programming system that are specific to evolving interest operators followed by the results of five experiments given in subsection 1.7.1.2.

1.7.1.1 Genetic Programming System Description

The genetic programming system's terminal set is a group of gray scale images. If the vision system works with color, the images use red, green and blue intensities or hue, saturation and intensity values. The genetic programming system's function set is {Neg, Abs, Square, ShiftL, ShiftR, ShiftU, ShiftD, +, -, *, /, Max, Min, Avg4x4, Sum4x4, PiN, AddN, MinN, MaxN}, with more detailed definitions given in figure 1.15. The functions operate on the whole image, performing the same operation on each pixel of the image. I_R is the resulting image generated by the function. The function's parameters are images using the name I. If a function has only one parameter, the

parameter name is just I. If a function has two or more parameters, a subscript distinguishes one parameter from another, e.g. I_i, where $i \in \{1, \ldots, 4\}$.

Function	Definition
Neg	$I_R(x, y) = -I(x, y)$
Abs	$I_R(x, y) = \|I(x, y)\|$
Square	$I_R(x, y) = I(x, y) * I(x, y)$
ShiftL	$I_R(x, y) = I(x+1, y)$
ShiftR	$I_R(x, y) = I(x-1, y)$
ShiftU	$I_R(x, y) = I(x, y+1)$
ShiftD	$I_R(x, y) = I(x, y-1)$
+	$I_R(x, y) = I_1(x, y) + I_2(x, y)$
-	$I_R(x, y) = I_1(x, y) - I_2(x, y)$
*	$I_R(x, y) = I_1(x, y) * I_2(x, y)$
/	$I_R(x, y) = I_1(x, y) / I_2(x, y)$
Max	$I_R(x, y) = \max\{I_1(x, y), I_2(x, y)\}$
Min	$I_R(x, y) = \min\{I_1(x, y), I_2(x, y)\}$
Avg4x4	$I_R(x, y) = \frac{1}{16}\sum_{-2\le i,j<2} I(x+i, y+j)$
Sum4x4	$I_R(x, y) = \sum_{-2\le i,j<2} I(x+i, y+j)$
PiN	$I_R(x, y) = \prod_{i=1}^{i=N} I_i(x, y)$, $N \in \{3, 4\}$
AddN	$I_R(x, y) = \sum_{i=1}^{i=N} I_i(x, y)$, $N \in \{3, 4\}$
MinN	$I_R(x, y) = \min\{I_i(x, y) \mid i \in \{1, \ldots, N\}\}$, $N \in \{3, 4\}$
MaxN	$I_R(x, y) = \max\{I_i(x, y) \mid i \in \{1, \ldots, N\}\}$, $N \in \{3, 4\}$

Figure 1.15 - Evolved interest operator function definitions.

$$fitness_{raw}(Ind) = \sum_{i=1}^{5}(U(Ind(I_i)) + \frac{1}{n}\sum_{p\in I_i}((Ind(I_i))(p) - (Moravic(I_i))(p))^2)$$

Figure 1.16 - Evolved interest operator fitness function.

The genetic programming system fitness function in figure 1.16 measures the difference between the output of an evolved interest operator, which is named Ind, and the output of the ideal Moravec interest operator, which is named Moravec. The fitness

function uses five different images as fitness cases to compute the fitness value of the evolved interest operator. The name I represents the fitness case images and the subscript i distinguishes one fitness case from another, e.g. I_i, where $i \in \{1, \ldots, 5\}$. The evolved interest operator Ind and the Moravec interest operator Moravec receive as input each fitness case I_i generating their respective output images $Ind(I_i)$ and $Moravec(I_i)$. The fitness function calculates the sum of the square of the difference between each corresponding pixel p of the output images $Ind(I_i)$ and $Moravec(I_i)$. Divide the sum of the square of the differences by the image's pixel count n adding the term $U(Ind(I_i))$. Ebner (1998) defined the term $U(Ind(I_i))$ as a large value if image $Ind(I_i)$ is uniform and 0 otherwise.

1.7.1.2 Experimental Results

Ebner (1998) performed five experiments using the parameters in figure 1.17. Each experiment used a different subset of the genetic programming function set according to the table in figure 1.15, where BASE = {Neg, Abs, Square, ShiftL, ShiftR, ShiftU, ShiftD, +, -, *, / Max, Min}. The five function subsets are: BASE, BASE $\cup$ { Avg4x4 }, BASE $\cup$ { Sum4x4 }, BASE $\cup$ { Sum4x4, Pi3, Add3, Max3, Min3 }, and BASE $\cup$ { Sum4x4, Pi3, Add3, Max3, Min3, Pi4, Add4, Max4, Min4 }.

The first experiment demonstrates that the BASE function set is adequate to generate an evolved interest operator that is close to the Moravec interest operator. The lack of the function Sum4x4 increases the amount of time the genetic programming system needs to evolve the interest operator because the system evolves its own version of Sum4x4 in four different places within the evolved interest operator. The second

experiment adds the function Avg4x4 to the BASE function set to help speed up the evolution process because Sum4×4 = Avg4×4 * 16. The third experiment adds the function Sum4x4 to the BASE function set, instead of Avg4x4, evolving an individual with the highest overall adjusted fitness value seen across all three experiments. Experiments four and five add additional functions to experiment three's function set. The response of the best individual of experiment 5 is very close to the response of the Moravec interest operator but there is a difference in the features detected by each operator. Ebner (1998) states that the application of non-local maxima suppression and a threshold operation which are not part of the fitness function cause the evolved operator to detect different features than the Moravec interest operator. This is only natural since the genetic programming system only evolves interest operators that approximate the response of the Moravec operator but does not necessarily detect the same set of features. An interest operator could be an integral part of a face tracking system locating key points that are potentially parts of a face.

Parameters	**Value**
Population size	4000
Generations	50
Crossover rate	85%
Mutation rate	5%
Reproduction rate	10%
Fitness cases (images)	5
Number of genetic programming runs	3

Figure 1.17 - Evolved interest operator experiment parameters.

1.7.2 Feature Detectors for a Face Tracking System

Guarda, Le Gal and Lux (1998) use genetic algorithms and genetic programming systems to create visual feature detectors for a face tracking system. The creation of the face tracking system is a two-stage process with the learning of features and detectors occurring in the first stage (subsection 1.7.2.1) and the integration of the detectors into a face tracking system occurring in the second stage (subsection 1.7.2.2). Subsection 1.7.2.3 contains a description of the results of the experiments to test the face tracking system.

1.7.2.1 The First Stage

The first stage consists of four components. The first component is storage for the various training examples used in the creation of features and detectors. The second and third components perform feature and detector learning, respectively, which comprise the bulk of the first stage. The last component collects the features and detectors into a library used by the second stage.

1.7.2.1.1 Training Images and the Feature/Detector Library

The user supplies most of the gray scale or color images used as training examples. The face tracking system can create example images for training as well. Light, shape and texture information are more difficult to extract from a gray scale image than a color image because much of the information blends together in a gray scale image. The intensity of light level variations in a color image helps to separate the

different pieces of information from one another giving preference to color images as training images. (R_0, G_0, B_0) represents the real color of an object but the color the observer sees (R, G, B) (defined in figure 1.18) is dependent on the set of light sources L, the light intensity I_p of light source p, and the angle θ_p between the object's surface normal and the light ray from light source p. Figure 1.19 lists the normalized values u, v, and w of the RGB color components, where u + v + w = 1. Figure 1.20 lists the redefined components u and v making them independent from light variations and object orientation. Define the input images in terms of their u, v, and gray-scale components. A database stores the training images until the learning components use them.

$$R = R_0 \sum_{p \in L} I_p \cos\theta_p \ , \quad G = G_0 \sum_{p \in L} I_p \cos\theta_p \ , \quad B = B_0 \sum_{p \in L} I_p \cos\theta_p$$

Figure 1.18 - Definition of a color's RGB components.

$$u = \frac{R}{R+G+B}, \ v = \frac{G}{R+G+B}, \ w = \frac{B}{R+G+B}$$

Figure 1.19 - The normalized RGB color components u, v, and w.

$$\begin{aligned} u &= \frac{R_0 \sum_{p \in L} I_p \cos\theta_p}{R_0 \sum_{p \in L} I_p \cos\theta_p + G_0 \sum_{p \in L} I_p \cos\theta_p + B_0 \sum_{p \in L} I_p \cos\theta_p} \\ &= \frac{R_0}{R_0 + G_0 + B_0} \\ v &= \frac{G_0 \sum_{p \in L} I_p \cos\theta_p}{R_0 \sum_{p \in L} I_p \cos\theta_p + G_0 \sum_{p \in L} I_p \cos\theta_p + B_0 \sum_{p \in L} I_p \cos\theta_p} \\ &= \frac{G_0}{R_0 + G_0 + B_0} \end{aligned}$$

Figure 1.20 - u and v definitions independent of light variation and object orientation.

The library component stores the masks and detectors either evolved by the system or provided by the user. The masks and detectors provided by the user allow the system to gain human expertise. The library supplies the masks and detectors needed by the genetic systems and the face tracking application.

1.7.2.1.2 Feature Learning as Masks

A mask is a two-dimensional array representing a feature. The value of each array element is in the range of [-5, 5]. An inner product of the mask at a particular spot in the image performs recognition of a feature at the position in the image. For each pixel of the inner product, a value in the range of [0, 1] specifies how close the pixel coincides with the mask. A genetic algorithm evolves the masks, storing them in the library component database. The library of masks provides the initial population of masks in the genetic algorithm. The genetic algorithm creates offspring using the standard reproduction operators of crossover and mutation. The crossover operator swaps equal sized rectangular regions between two parent masks to form two new offspring masks. The mutation operator alters the values in a rectangular region within the mask. Random means chooses the size of the rectangular region. The fitness of mask M in image I is an error measure where a score of 0 denotes a perfect mask. A window W encloses the feature in image I and $max_w M$ is the maximum response of mask M within window W. $M(pixel_{ij})$ computes the response of mask M to the pixel in image I at coordinates (i, j). The genetic algorithm fitness function (see figure 1.21) counts the number of pixels outside window W in image I, I –W, in which the mask M's response, $M(pixel_{ij})$, is greater than the mask M's maximum response inside of W, $max_w M$. A mask M's total

fitness is the sum of its fitness for each image in the set of training images. The feature learning component works concurrently with the detector learning component. Later runs of the detector learning component use the results of current runs of feature learning.

$$fitness_M(I) = \sum_{pixel_{ij} \in I - W} greater(M(pixel_{ij}), \max_W M)$$

Figure 1.21 - Mask fitness function.

1.7.2.1.3 Detector Learning

Detectors bring together several feature masks using fuzzy logic rules to specify logical relationships between the masks. Genetic programming techniques evolve detectors with the fuzzy logic functions "NOT", "OR", "AND" and "EXIST" forming the function set and the library of masks forming the terminal set. The detector focuses on a particular point in the image with a window surrounding the point's local neighborhood divided into nine equal sub-windows. The function "EXIST" determines if the feature of a mask is present within a particular sub-window of the detector. The genetic programming system uses the standard genetic operators found in any typical genetic programming system to create detector offspring.

The genetic programming fitness function (see figure 1.22) calculates a detector's error with a score of 0 denoting a perfect detector. The genetic programming fitness function uses the same training examples used by the genetic algorithm fitness function. There are three factors that characterize a good detector: sensitivity, specificity and efficiency. Each term in the fitness function, defined as an error value, represents a different aspect of the detector's fitness. The sensitivity term $f_{sensitivity}$ (see figure 1.22)

specifies how well the detector senses the center of the object. A smaller $f_{sensitivity}$ value is better because it means the detector is more sensitive to the center of the object. The specificity term is another indicator of the detector's sensitivity to the center of the object by concentrating on the maximum response of the image's pixels outside of the object's center. A better detector has a specificity value, $f_{specificity}$ (see figure 1.22), closer to 0 because it signifies that the detector focuses more on the pixels inside the object center. The size of the detector's parse tree specifies the efficiency of the detector (see figure 1.22). A smaller parse tree defines a more efficient detector. The detector library stores the fittest detectors at the end of each run, for use in the second stage.

Component	**Equation**
Detector fitness function	$f_{detector} = f_{sensitivity} + f_{specificity} + f_{efficiency}$
Detector's sensitivity	$f_{sensitivity} = 1 - \text{response}(\text{ObjectCenter})$
Detector's specificity	$f_{specificity} = \max_{pixel \in \text{Image-ObjectCenter}}(\text{response}(\text{pixel}))$
Detector's efficiency	$f_{efficiency} = \text{depth}(\text{Detectors\_Parse\_Tree})$

Figure 1.22 - Detector's fitness function.

1.7.2.2 The Second Stage

The second stage integrates the detectors into a face recognition system and a system to provide new training examples for the first stage. A face tracking application is a complex and difficult process because the face's environment can vary in lighting and background conditions. Six processes comprise the second stage: a supervisor, image acquisition and processing, the detectors, tracking, camera control, and the first stage

(mask and detector learning). The supervisor process controls the communication between and execution of each process of the system. The supervisor determines the best detector to use for the current image in the face tracking image sequence as well as adding the current image to the training examples for the first stage.

1.7.2.3 Experimental Results

The face tracking application experiments use a 100-image sequence to learn detectors for the eyes where ten percent of the images end up as training data. The genetic algorithm portion of the learning process employed a mask population of 500 individuals evolved over 50 generations. The genetic programming system portion of the learning process employed a detector population of 600 individuals evolved over 300 generations. Ten runs of the learning process use only color images and ten more runs of the learning process use only gray scale images. Although the color image run producing the best detector took longer to converge than the gray scale image run producing the best detector; the color image run evolved a better detector than the gray scale image run. The application of the best detector of all twenty runs to each of the 100 images measures the detector's response to the eyes in each image. The response's value ran from a value of 0, signifying the detector did not respond to the eyes, up to a value of 1, signifying the detector's maximum response to the eyes. As expected, the best detector returned perfect results for the training sequence images. For the non-training images, the best response is always on the eyes with the minimum response value recorded around 0.88. When added to the tracking system, a kalman filter helps to reduce the errors of the detector.

This subsection investigated the development of an evolutionary system to evolve a face tracking application. The evolutionary system contains a certain level of parallelism. The processes of the second stage work in tandem. In addition, the feature learning and detector learning components of the first stage execute in relative independence of each other. The face tracking application is a sequential program. Subsection 1.7.3 illustrates the integration of parallelism into feature detection through the use of autonomous agents to detect features in an image.

1.7.3 Autonomous Agents to Detect Features

The use of parallelism dramatically increases the detection of features and objects in a two-dimensional image. In their initial research, Liu, Maluf and Tang (1997) developed evolutionary autonomous agents working in parallel that travel through an image identifying the borders of closed regions. Liu and Tang (1999) adapted the evolutionary autonomous agents from their earlier work to perform image segmentation. Each subsection contains a description of the evolutionary algorithm and the results of the experiments conducted on the algorithm.

1.7.3.1 Detecting Borders of Closed Regions

A two-dimensional 8-connected grid of pixels stores the image with each agent dwelling in one of the pixels. At the beginning of each time step of the evolutionary algorithm, the agent calculates the density distribution (see figure 1.23) of all the pixels in the agent's local area. Note the usage of s^0t^0 in figure 1.23 instead of its equivalent value 1. The authors do this to stress the point that, in future research, the density calculation

may be the summation of higher-ordered moments involving s and t. I(i,j) represents the gray-scale value at pixel (i,j), r denotes the radius of local region, and δ designates a predefined positive constant. The agent uses the density distribution value to determine the next action the agent carries out. If the density distribution value falls within the agent's interval λ = [u, v], where u ≤ v, the agent reproduces a random number of offspring placed randomly within the agent's local neighborhood.

$$D^{r}_{I(i,j)} = \sum_{s=-r}^{r} \sum_{t=-r}^{r} \{s^0 t^0 \mid \| I(i+s, j+t) - I(i,j) \| < \delta\}$$

Figure 1.23 - Density distribution for the agent at pixel (i, j).

The offspring inherit all the properties of their parent except the age. If the density distribution value does not fall within the agent's interval λ then the agent moves from its current pixel in some arbitrary direction to another pixel according to the agent's deviation vector. The agents have a time constraint; the number of generations (termed "age") that they are active before they cease to perform and removed from the population. The fitness of an agent is the spatial deviation of its current location from the already selected/pre-selected critical feature regions.

After performing its action, each agent calculates its fitness based on its spatial deviation from already selected or pre-selected (by user) sensitive locations in the image. A high fitness value indicates the agent is closer to a sensitive area while the opposite is true for a low fitness value. Evolution in this system uses strictly two types of mutations, an r-mutation or a d-mutation. Agents with a high fitness value undergo an r-mutation, which changes the value of the agent's radius. The new radius value, r^{n+1}, replaces the current radius value, r^n, when $S(r^{n+1}) < S(r^n)$, where S(…) is the regional reproduction

rate cost function. Agents with a low fitness value undergo a d-mutation, mutating the agent's ability to displace itself from a sensitive region, which introduces error values into the agent's deviation vector. Each agent keeps its own age value. If the agent ages a certain number of time steps before reproducing then the agent ceases to exist and vanishes from the image.

The experiments used a 150 by 150 pixel 256-gray-scale image dispersing 400 agents into the image with the agent's life span set to 5, λ interval = [2, 6], radius r = 1 and the pre-defined constant $\delta = 45$ for the density distribution function of figure 1.23. Initially, most of the agents were not close enough to detect the closed regions but given a few time steps were able to close in on a nearby region's border. All of the regions in the image were detected. The experimentation indicated that the initial number of agents does not affect the total number of agents at the end. Rather, initial population size affects d-mutation and death rates. A higher initial population tends to have faster convergence rates. The experimentation found that increased life span increases the radius (domain) of fitness distribution causing more agents for selection of the next generation. However, the more agents selected for r-mutation, the longer time experienced by an agent to travel nearer to a sensitive region.

1.7.3.2 Performing Image Segmentation

Liu and Tang (1999) built upon the work done in their previous paper (Liu, Maluf and Tang, 1997) to design autonomous agents that perform image segmentation. A two-dimensional image S stored in an array of size U×V contains one or more homogeneous segments with an initial group of autonomous agents randomly distributed into the image.

Each autonomous agent has and evolves two behaviors that allow the agent to travel through the image to try to find and label pixels of a particular homogenous segment. The first behavior breeds offspring agents and places them into pixels within the agent's local region. The second behavior diffuses the agent in a direction that hopefully contains the agent's particular homogeneous segment. Each vector within the agent represents a behavior. $P_\omega(\Omega)$ represents the breeding vector where Ω is the set of N possible breeding direction sectors. $Q_\theta(\Theta)$ represents the diffusion vector where Θ is the set of N possible diffusion direction sectors. Each element in a vector represents the probability of successful breeding or diffusion occurring in that particular direction (see figure 1.24).

$$P_\omega(\Omega) = [p_1, p_2, \ldots, p_N], \text{ where } \sum_{\omega=1}^{N} p_\omega = 1 \text{ and } p_\omega \in [0,1].$$
$$Q_\theta(\Theta) = [q_1, q_2, \ldots, q_N], \text{ where } \sum_{\theta=1}^{N} q_\theta = 1 \text{ and } q_\theta \in [0,1].$$

Figure 1.24 - Definition of the breeding and diffusion vectors.

$$f(\tilde{\alpha}) = \begin{cases} 1 - \dfrac{\#\,of\ steps\ before\ breeding}{life\ span\ of\ \tilde{\alpha}}, & if\ \tilde{\alpha}\ finds\ a\ triggering\ stimulus. \\ -1 & otherwise. \end{cases}$$

Figure 1.25 - Agent fitness function calculating the fitness of agent $\tilde{\alpha}$.

At the beginning of each evolutionary algorithm time step, each active agent α_i updates their breeding and diffusion vectors, evolving a better agent. First, locate the successful agents from among agent α_i's parent and siblings. A successful agent has a high fitness value, a fitness value greater than 0, taking the least number of steps to find the homogeneous segment. The fitness of an agent bases its value upon the number of

steps an agent takes to find a homogeneous segment (see figure 1.25). Second, adjust the values of agent α_i's breeding and diffusion vectors using the corresponding vector values from all the successful agents found.

Each active agent performs one of the two behaviors after updating the vectors of all active agents. The terms of the three criteria values: relative contrast, regional mean and regional standard deviation mathematically define the homogeneous segments. The criteria tell the agent if the pixel the agent currently inhabits is part of the homogeneous segment. Relative contrast specifies the number of pixels within a radius *r* of the agent's current location that have intensity close to the intensity of the pixel at the agent's current location. The regional mean is the average intensity of all the pixels within a radius *r* of the agent's current location. The regional standard deviation is the standard deviation of the intensity of all the pixels within a radius *r* of the agent's current location. The precise definitions of these three criteria depend on the type of detected homogeneous segment. If the agent's inhabited pixel is a part of the homogeneous segment, the agent breeds offspring agents, labels the pixel as found, and becomes inactive. The parent agent breeds a finite number of offspring agents, set them as active, and sends them in different directions within a certain radius of the parent agent. The parent agent's breeding vector determines the direction a parent agent places its offspring agent. Creation of an offspring agent's vectors comes from the vectors of its parent agent and successful sibling agents. If the inhabited pixel is not a part of the homogeneous segment, the agent diffuses (travels) to another nearby pixel in an arbitrary direction searching for the homogeneous segment. When an agent needs to choose a direction to travel in search of

a homogeneous segment, the agent uses its diffusion vector. The evolutionary algorithm (see figure 1.26) executes until the number of active agents reaches 0.

Images usually contain more than one type of homogeneous segment. Therefore, there is one type of agent designed for searching each type of homogeneous segment. Liu and Tang (1999) experimented with a 612×792 sized brain scan image comprised of four different homogeneous segments. The experiment began by depositing 4 groups of 500 agents each, with each group searching for a different segment. Detection of all four segments needs approximately 70 time steps. Various classes of segments in numerous images provide data to several other experiments. Liu and Tang (1999) uncovered three characteristics the evolved agents developed in the various experiments: 1) the agents have simple and well-defined dynamic behaviors, 2) the agents sense and react to only a few number of neighboring pixels at any one time, and 3) the agents adjust and focus their attention on the areas in the image expected to contain homogeneous segment pixels. Finally, graphing of the number of active agents in the image over time measures the algorithm's computational cost. In most experiments, the number of active agents peaks early, by time step 10. The number of active agents decline with the labeling of more and more homogeneous segment pixels over time. The next section examines a mechanism to evolve non-linear image processing filters.

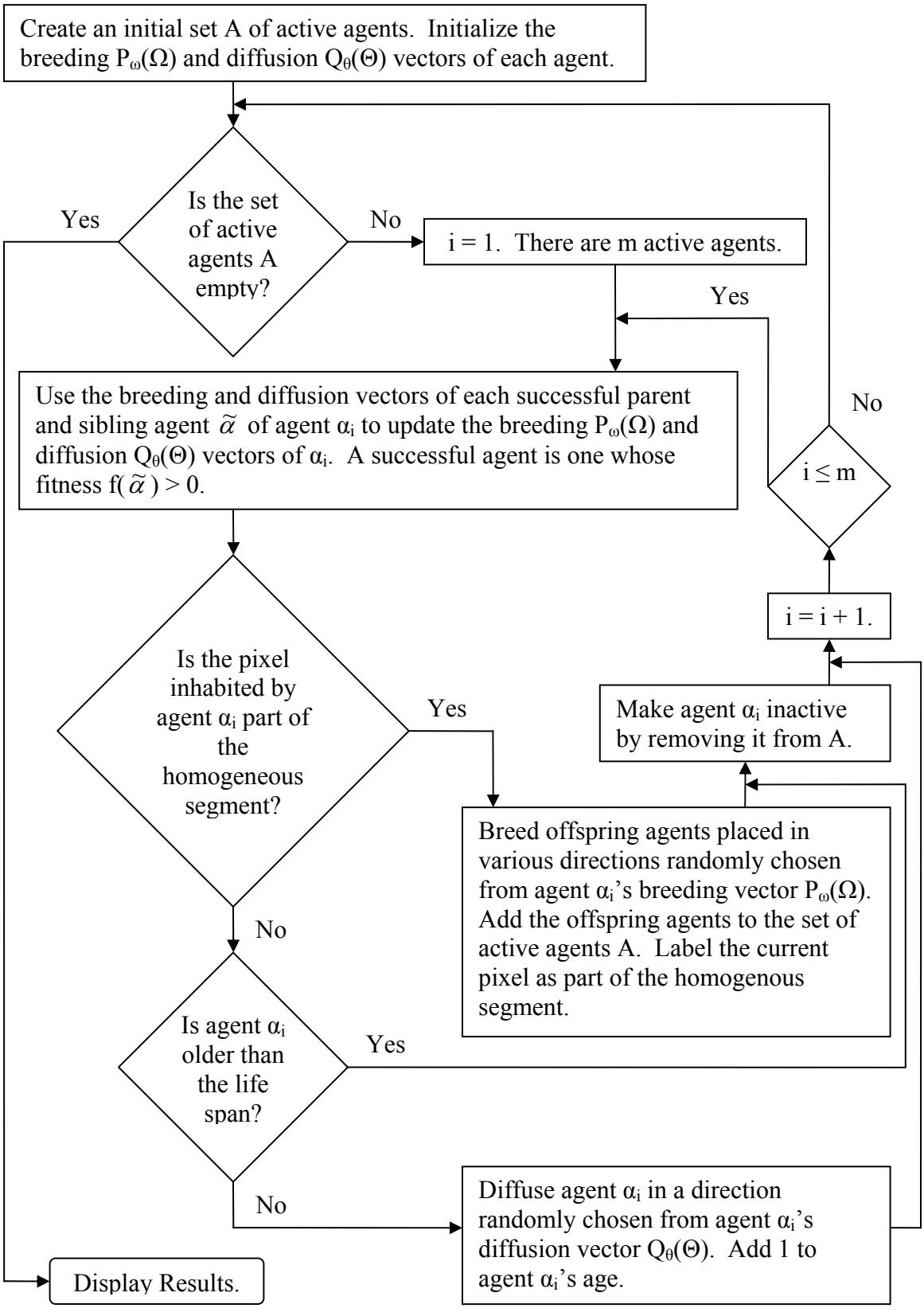


Figure 1.26 - Autonomous Agent Image Segmentation Evolutionary Algorithm.

1.8 Non-Linear Filters

Frederick Waltz (1994a) developed the **s**eparated-**k**ernel **i**mage **p**rocessing using finite **s**tate **m**achines (SKIPSM) architecture to provide a technique to create fast and minimal implementations of common image processing operations such as binary morphology (Waltz and Garnaoui, 1994a), Grassfire Transform (Waltz and Garnaoui, 1994b), binary template matching (Waltz, 1994b), and grey-level morphology (Waltz, 1994c). A two-dimensional grid of pixel values having I rows and J columns is the structuring element that defines the image processing operation. Placing the structuring element on the image, each structuring element and image pixel value pair performs a pixel test (q_{ij}, i = 1, …, I, j = 1, …, J). The overall output value of the image processing operation is a function $F(q_{11}, \ldots, q_{1J}, q_{21}, \ldots, q_{2J}, \ldots, q_{I1}, \ldots, q_{IJ})$ of the combination of all the pixel test values. The row function R represents the result of combining the pixel test values for one row. The column function C represents the overall result of combining the results for all the rows. If the image processing operation is separable then a group of row operations R followed by a column operation C (see figure 1.27) defines the overall function for the operation.

$$F(q_{11}, \ldots, q_{1J}, q_{21}, \ldots, q_{2J}, \ldots, q_{I1}, \ldots, q_{IJ}) = C(R(q_{11}, \ldots, q_{1J}), R(q_{21}, \ldots, q_{2J}), \ldots, R(q_{I1}, \ldots, q_{IJ}))$$

Figure 1.27 - Image processing operation function defined in terms of row and column functions.

This separation of the image processing operation is the basis for the SKIPSM architecture. According to the SKIPSM architecture, a combination of two finite state machines, a row machine pipelined into a column machine (see figure 1.28) implements

the image processing operation. The row machine performs the role of the row function and the column machine performs the role of the column function.

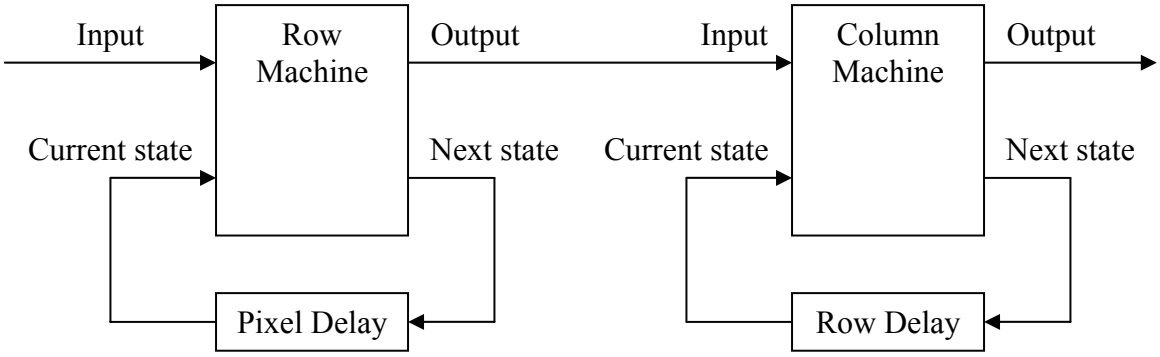


Figure 1.28 - Basic SKIPSM architecture.

Constructing the row machine involves examining all the rows of the structuring element, with each row forming a pattern, and labeling each distinct row. The row machine must recognize each distinct row, including a null row pattern. The null row pattern represents all other possible row patterns except the distinct row patterns of the structuring element. The null row pattern also has a label. The set of distinct row labels forms the output alphabet of the row machine. The row machine processes each row of an image from rows 1 to I, receiving as input each image pixel value of a row in sequence from the pixel in column 1 to the pixel in column J. Upon processing the last pixel of the image row, the row machine outputs the closest matching structuring element row label or the null row label if the image doesn't match any structuring element row. The pixel delay of the row machine keeps the next state calculation in synch with the next input pixel value. The column machine receives row labels as input from the row machine. The row delay of the column machine keeps the next state calculation in synch with the next input row value. The column machine recognizes the particular sequence of row

patterns specific to the structuring element of the image processing operation. According to the description above, the column machine terminates once it receives and processes the last row of the structuring element for the current position of the structuring element in the image.

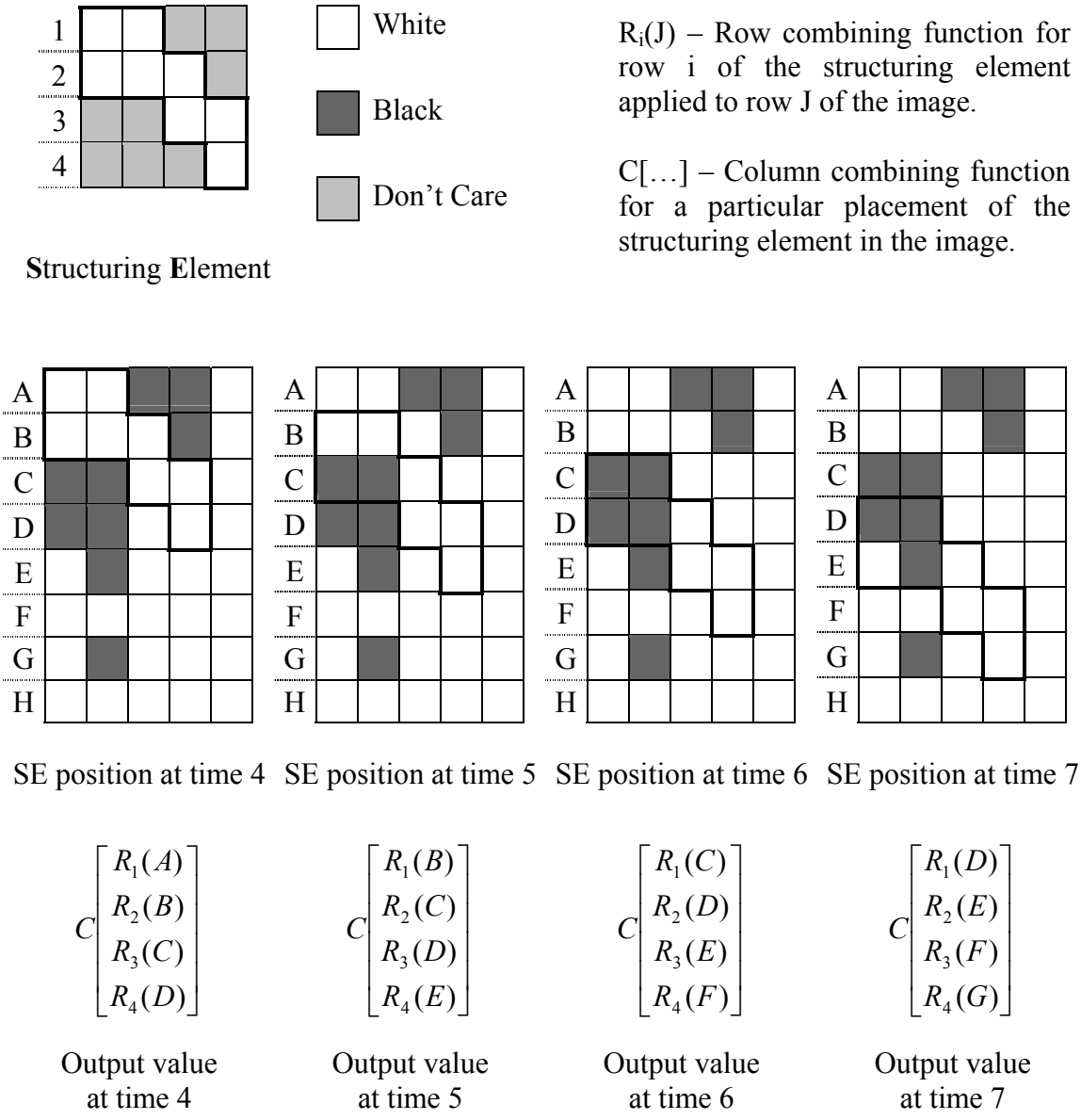


Figure 1.29 - SKIPSM machine computation for four time steps.

The SKIPSM machine traverses an image from the image's upper left hand corner to its lower right hand corner to find each place in the image that matches the SKIPSM machine's structuring element, see figure 1.29. This is possible because of the pipelined nature of the SKIPSM machine. Figure 1.29 illustrates a four row structuring element at four consecutive positions in an image with the output value calculations for each position listed at the bottom of figure 1.29.

The value 0 represents a black pixel and the value 1 represents a white pixel in the binary image. The value 1 represents a white pixel and the value 0 represents a "don't care" pixel in the binary structuring element. The "don't care" value means that the pixel at that relative position within the image can be white or black. Starting from row A downwards, each row of the image feeds into the SKIPSM machine's row machine only once and in consecutive order. In addition, the column machine remembers the partially completed column output value calculations. Assume that the row machine processed row D and sent its output value to the column machine. Each row in the image tests against each row of the structuring element, for example, see row D in the column combining functions of figure 1.29. A simple examination of rows 1 and 2 of the structuring element shows how each row in the image, fed into the SKIPSM machine's row machine only once, tests against each row of the structuring element. According to row 1, the first two pixels must be white and the remaining two pixels are either white or black. Row 1 represents four possible row patterns (see figure 1.30). According to row 2, the first three pixels must be white and the remaining pixel is either white or black. Row 2 represents two possible row patterns (see figure 1.30). When a row of the image feeds into the row machine and the row machine outputs a 2, the column machine

assumes that the image row matches both row 1 and row 2 of the structuring element. In summary, Waltz (1994a) constructed an automated method of generating a SKIPSM row and column machine for a specific image processing operation according to the operation's structuring element. Next, the final section of this chapter examines several applications of evolutionary computation to optical character recognition.

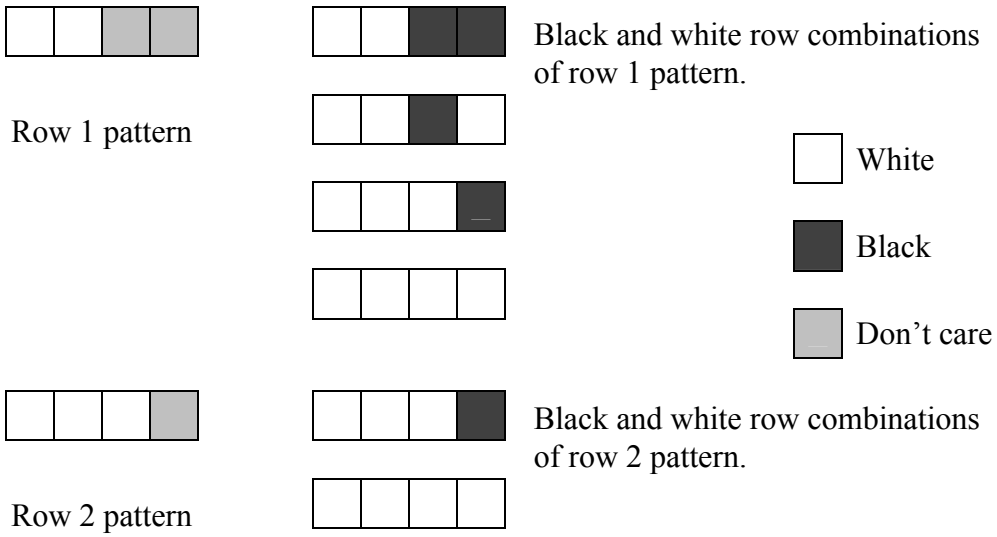


Figure 1.30 - Black and white row combinations of row patterns.

1.9 Optical Character Recognition

Optical character recognition can benefit from the use of evolutionary computation in various ways. This section illustrates several such uses taken from the recent literature. The first evolutionary computation use (Andre, 1994) develops a genetic programming system to evolve new and existing rules for an optical character recognition system. The genetic programming system can utilize, for the initial population, code written by humans. The final set of evolved rules was successful at

recognizing large character sets in multiple fonts and sizes. De Stefano, Cioppa and Marcelli (2002) also use genetic programming to evolve a set of rules used by a pre-classification system to describe the shape resemblance between characters in a handwritten character recognition system. The rules utilize the feature values from a character shape description scheme specified by a feature vector.

The next evolutionary computation use evolves character recognition feature detectors or agents. Dellaert and Vandewalle (1994) developed a genetic algorithm to evolve feature detectors to recognize digits of any font and size. Kharma, Kowaliw, Clement, Jensen, Youssef, and Yao (2004) developed CellNet, a cooperative co-evolution genetic algorithm to evolve agents to recognize characters. The genetic algorithm makes use of a new genetic operator called Merger that creates a new agent from a combination of two existing agents. The genetic algorithm utilizes two populations. The first population is the hunters, pattern recognizers, where features define the hunter's structure from a finite set of features. The other population is the prey, the set of training images. A set of algorithms defines the behavior of each population, the environment. A standard genetic algorithm without the Merger operator tests the performance of the CellNet system on two sets of handwritten characters. The CellNet system converges faster to a highly accurate character recognition agent than the standard genetic algorithm. Initially, the authors only evolved the population of hunters but modified the CellNet system for competitive co-evolution where the prey evolves as well through the use of camouflage functions to produce hunter agents that are even more reliable than the first set of evolved hunters.

The next evolutionary computation use evolves the best set of features for character recognition. Anderson, Asbury, Gaborski and Tilley's (1995) handwritten character recognition algorithm utilizes 1500 features extracted from the image. Some features overlap significantly with other features while certain features are ineffective. Anderson, Asbury, Gaborski and Tilley (1995) devised a genetic algorithm to converge on the most effective subset of features to use in their handwritten character recognition algorithm. The authors were successful in their goal, finding the 300 most effective features.

Oliveira, Benahmed, Sabourin, Bortolozzi, Suen (2001) also utilize a genetic algorithm to converge on the most effective subset of features for use in their handwritten character recognition system. A set of features forms a classifier to recognize the handwritten digits in an image. A genetic algorithm evolves subsets of features in order to find the smallest subset of features which creates a classifier with the lowest error rate. The authors used a "simple" genetic algorithm which evolves the most effective feature subset in one run. The authors also used an iterative genetic algorithm which evolves the most effective feature subset in several iterations of runs. The initial run begins with the full set of features, converging to a feature subset which has a classifier error rate close to zero. Each succeeding run of the genetic algorithm utilizes the best feature subset from the previous genetic algorithm run. The authors discovered that four runs of the genetic algorithm found the best feature subset. The experiments tested both types of genetic algorithms. Each type of genetic algorithm found a good subset of features. The simple genetic algorithm tended to examine a more narrow area of the search space while the iterative genetic algorithm examined a broader area of the search space. In the end, the

authors found the feature subset of the simple genetic algorithm worked best in their handwritten character recognition system. Finally, Sural and Das (2001) utilize a genetic algorithm to converge on the most effective subset of features to use in a neuro-fuzzy optical character recognition system.

The next evolutionary computation use evolves the best of weights for character recognition. Berlanga, Garcia-Herrero, Molina, Besada, and Portillo (2002) utilize an evolutionary algorithm to converge on the most effective set of weights of a Euclidean distance measure for use in successfully recognizing aircraft tail numbers in an optical character recognition system.

Cha and Tappert (2003) utilize gradient, structural, and concavity (GSC) features to recognize characters. A set of weights applied to the feature set provide the character recognition system with a method to compute the hamming distance between a "character" in an image to the closest character in the character database. The authors use a genetic algorithm to optimize the set of weights for character recognition.

Lin, Wang and Yeung (2005) developed a genetic algorithm to find the optimal weights for combining a set of classifiers for a handwritten Chinese character recognition system. The resulting combinatory classifiers found by the genetic algorithm performed better classification than the original component classifiers used in creating the combined classifiers.

In the next evolutionary computation use, Pan, Ye and Zhang (2005) developed a hybrid method for automobile license plate character recognition. A two stage recognition process identifies the characters after character extraction from the image. The first stage uses statistical methods to identify the characters while the second stage

uses structural methods to recognize the indistinguishable characters from stage one. A genetic algorithm tunes the system's parameters used during the two stages of character recognition.

In the final evolutionary computation use, Wei, Ma and Jin (2005) created a genetic algorithm to segment handwritten connected Chinese characters. The individuals in the genetic algorithm population are line segments dividing the two Chinese characters. The line segment exists only within a narrow horizontal band in the middle of the image. The genetic algorithm fitness function is the mixed Gaussian function that makes use of an eight valued feature vector extracted from the image with the line segment already in place. The author's genetic algorithm has an 89% accuracy rate. The next chapter states the first dissertation problem along with the means and methods to address this problem.

Chapter 2 The Signature Recognition Problem

The signature recognition problem attempts to find a match between a given unknown handwritten signature to one of a database of known handwritten signatures. Scan the handwritten signature into an image file, a digitized grid. Apply a standard thinning algorithm to the signature to create a version of the signature in which the lines and curves of the signature are one pixel wide. Think of the one pixel wide version of the signature as a path. Evolving agents (finite state machines) dedicated to particular paths is a means to recognizing handwritten signatures.

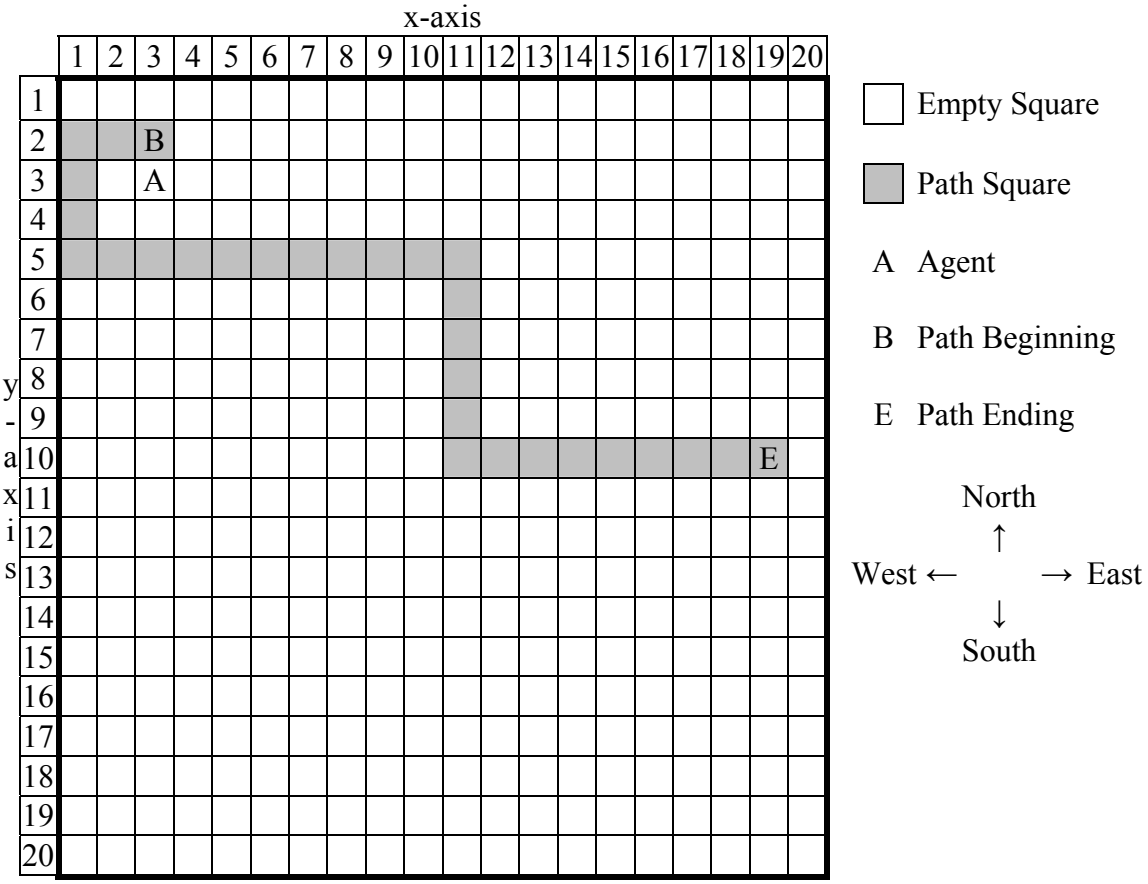


Figure 2.1 - A 20 x 20 grid containing a path and an agent.

This dissertation determines the feasibility of a genetic algorithm evolving an agent to consume an entire path. The world the agent occupies is a 200 by 200 grid of

squares. The path is a contiguous sequence of squares within the grid. Figure 2.1 illustrates a 20 by 20 grid containing a path and an agent. One end of the path designates the path's beginning and the other end designates the path's ending. Place the agent in one of the empty grid squares immediately adjacent to the beginning of the path, ready to take actions based on the input received and the path strategy inherited by the agent. The contents of the grid square immediately in front of the agent according to the current direction the agent faces: north, south, east, or west, determines the agent's input. The direction north corresponds to decreasing the y coordinate, the direction south corresponds to increasing the y coordinate, the direction west corresponds to decreasing the x coordinate and the direction east corresponds to increasing the x coordinate. The agent facing one of the edges (cliff) of the grid generates an input of "C", a grid square containing a path square generates an input of "S", or an empty grid square generates an input of "E". The agent uses the input value to determine the agent's next action. There are three possible actions: "A", "L", or "R". The action "A" has the agent move forward from the grid square it is currently occupying to the grid square directly in front of itself. The action "L" has the agent first changing direction by turning left (rotating 90 degrees) and then moving forward from the grid square it is currently occupying to the grid square directly in front of itself. The action "R" has the agent first changing direction by turning right (rotating -90 degrees) and then moving forward from the grid square it is currently occupying to the grid square directly in front of itself. The agent moves forward one grid square at the end of each action. But, if the agent faces a grid edge, the agent can't move forward one grid square. To handle this situation, the agent doesn't move forward.

The program written for this problem doesn't simply run the genetic algorithm once with a specific set of parameters for a given path. The program executes the genetic algorithm a set number of times for each given path. The discussion of the program and its genetic algorithm begins by reviewing the components and tools used by the program and genetic algorithm. To initiate this discussion, the next section states a few reasons why a good random number generator is essential for any genetic algorithm. Section 2.2 provides a description of the data structures used to represent the environment and behavior of the agent. Section 2.3 provides a detailed explanation of the fitness function used to compute the fitness of an agent. Section 2.4 discusses all of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation. Section's 2.5 and 2.6 give a description of the signature recognition genetic algorithm and the signature recognition program, respectively. The landscape of this problem's search space is essential for understanding the results of the signature recognition program with the discussion of the landscape generation program provided in section 2.7.

2.1 Random Number Generator

A good random number generator is an essential tool of a genetic algorithm because a genetic algorithm run requires several hundred thousand or million random numbers. A random number should be a number not predicted in advance of its creation. A random number generator written in a computer program employs an algorithm that gives the appearance of generating a random number. But, an algorithm is a sequence of

actions executed in a particular order, a predictable entity. Therefore, a random number generator creates a pseudo random number.

A random number generator starts with an initial seed value, executes a sequence of operations to manipulate the seed value and generates the pseudo random number. The current pseudo random number is the seed value for the next call to the random number generator. The design of a good random number generator addresses the following two problems, (Ladd, 1995) page 38:

1. The generated random number seeds the next call of the random number generator. Eventually, the new seed value will cycle back to the initial seed value. The longer the period between repetitions of the initial seed value, the more the sequence of pseudo random numbers appears random.

2. Random number generators that always create odd numbers or numbers with the same first digit are worthless because these cases are predictable.

The linear congruential method is the simplest technique to generate a random number making use of two basic mathematical operations. The rand() function found in most ANSI C compilers implements this method. The rand() function is inadequate because the range of values it generates is small, between 0 and 32767, and rand()'s repetition cycle is short, several thousand values. A genetic algorithm needs a random number generator with a repetition cycle of 100,000 to 1 million values for a typical run.

The uniform deviates method is a technique that generates a floating-point random number between 0.0 and 1.0. It is equally likely to generate each value in this range. The minimal standard is the basic model.

In the article “Efficient and Portable Combined Random Number Generators” in *Communications of the ACM*, Paul L'Ecuyer (1988) discussed several techniques for creating random number generators with a long period that produce reliable random numbers. L'Ecuyer (1988) accomplishes this by combining 2 different versions of the minimal standard. L'Ecuyer (1988) uses approximate factorization in his algorithm to remove correlation in the lower order bits of the result. L'Ecuyer’s (1988) algorithm has a period of 108, which is inadequate for genetic algorithms. But, L'Ecuyer’s (1988) generator can have a period of 2.3E1018 if combined with other types of random number generators and given a selection of excellent factors. This signature recognition program uses the random number generator coded by Ladd (1995). Section 4.2 in chapter 4 provides a detailed description of the two methods to generate integer and floating point random numbers for the signature recognition program. The next section provides a description of all of the structures required by the signature recognition problem.

2.2 Genetic Algorithm Structures Required for Signature Recognition

The three main structures required for the genetic algorithm are the grid, the agent and the path. Outlined in this section is a description of the implementation of each of these structures. A 200 row by 200 column two-dimensional array implements the grid. Each element of the array is a linked list of integers to allow multiple path squares to occupy a grid square. An empty linked list represents an empty grid square and a non-empty linked list represents a grid square containing one or more squares of the current path stored in the grid. Integer i in the grid represents the i^{th} path square in the grid. The

integers in each grid square's linked list are kept in ascending order from the head node to the tail node.

States	Inputs S	Inputs E	Inputs C
0	L/23	A/5	L/25
1	L/22	R/19	A/1
2	L/5	A/14	A/18
3	R/18	R/19	A/30
4	A/18	A/28	A/21
5	R/18	L/0	R/26
6	L/4	R/24	L/1
7	A/24	L/4	R/23
8	A/26	L/4	A/28
9	L/18	A/26	L/18
10	L/1	R/0	A/15
11	A/26	R/2	R/0
12	A/24	R/14	R/23
13	R/13	R/22	L/7
14	A/19	L/23	R/26
15	R/8	R/14	R/0
16	L/5	A/21	A/1
17	A/18	A/24	R/17
18	A/18	R/17	R/30
19	R/13	L/11	L/28
20	A/3	R/17	L/14
21	R/20	A/28	R/23
22	R/22	A/17	A/0
23	A/2	R/28	R/12
24	R/12	R/23	A/7
25	L/12	L/15	L/10
26	R/4	A/21	R/14
27	L/13	L/2	R/0
28	A/16	R/7	L/12
29	A/15	L/6	L/31
30	A/7	R/22	R/11
31	L/2	A/0	R/10

Figure 2.2 - An example of an agent.

Each agent has 32 states and 3 inputs defining its behavior because the agent's 3 inputs correspond to the 3 inputs (Square, Empty, Cliff) it can receive from its environment (see section 2.0) and each of the outputs contained in the agent correspond to one of the 3 actions (Advance, Left, Right) it can perform (see section 2.0). A two-dimensional array implements an agent with figure 2.2 providing an example of an agent. The rows of the array correspond to the agent's states and the columns of the array correspond to the agent's inputs. Each element in the array stores an "output/next state" pair. The output specifies the action the agent carries out and the next state indicates the state the agent enters after executing the action. The agent keeps track of its current state via a variable called "state" containing the agent's current state number. Section 4.3 in chapter 4 provides a detailed description of the actual implementation of an agent.

The experiments of the signature recognition program operate on one of two sets of 10,000 paths. The non-crossover paths are the first set of 10,000 paths where no part of a path can crossover any other part of the path. The crossover paths are the second set of 10,000 paths where any part of a path can crossover any other part of the path. The path creation program creates each set of paths beforehand storing the set in a group of data files. The main path creation method creates a path one leg at a time. A leg of the path represents one of the line segments of the path with the legs of the path stored in the order of creation. The direction of each new leg is perpendicular to the direction of the previous leg. Two values represent the leg, its length and the direction to move, in order to re-create the leg in the grid. After path creation, the main path creation method writes, to the data file, the sequence of integers representing the path with each path kept on a separate line. Section 4.4 in chapter 4 provides a detailed description of the methods to

create paths for the data sets of the signature recognition program. The next section discusses the fitness functions used in the various signature recognition program experiments.

2.3 The Fitness Function

The fitness function of a genetic algorithm quantifies the "success" of an individual for a given task. For the signature recognition program, this measures the fitness of an agent as to how much of the path the agent traverses and consumes. Therefore, an agent's fitness can have a maximum value equal to the length of the path. There are two different fitness functions for the experiments of the signature recognition program. The first fitness function, computeFSMFitness, measures how much of the path the agent consumes regardless of the order of path square consumption.

The computeFSMFitness fitness function keeps track of the agent's fitness value, the agent's current state, the number of actions the agent executes, the current direction the agent faces, and the current square the agent occupies in the grid. The function initializes the agent's fitness value and number of actions to zero. The start state for every agent is state 0. Therefore, the function initializes the agent's current state to zero. The function places the agent in the grid at the designated start square which is immediately next to the beginning square of the path and sets the agent's direction to the designated starting direction. The agent performs the following steps until one of two termination conditions is met. Step 1: the agent examines the contents of the grid square it currently occupies. If the grid square's linked list contains at least one path square not yet consumed, the agent consumes the path square by increasing its fitness value by one

and setting the grid square's linked list pointer to point to the next node in the linked list. Step 2: the agent determines if an edge of the grid or a grid square is directly in front of itself in the direction the agent faces. If the agent faces an edge of the grid, the agent receives an input value of "C" for cliff. If the agent faces a grid square, the agent receives an input value of "E" if the agent consumed all of the grid square's path squares, or the agent receives an input value of "S" if the grid square's linked list contains at least one path square not yet consumed. Step 3: the agent retrieves from its finite state table the "output/next state" entry at the intersection of the current input value and current state. Step 4: the agent performs the action associated with the retrieved output value. The agent increases its number of actions executed by one. Step 5: the agent sets its current state to the retrieved next state value. Step 6: If the agent's fitness value is equal to the length of the path in the grid or the number of actions the agent executed is equal to five times the length of the path in the grid, the fitness function terminates returning the agent's fitness value; otherwise the fitness function returns to step 1.

The second fitness function, computeFSMPCFitness, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. A full contiguous order means the agent consumes each path square in order from the beginning path square to the end path square. An agent can also consume the path in a partial contiguous order. A partial contiguous order means the agent consumes segments of the path in a full contiguous order but the agent consumes the path squares between segments out of order. The agent consumes path squares within a segment in a full contiguous order. The computeFSMPCFitness fitness function requires labeling each path square i with the value of integer i. In addition, computeFSMPCFitness also keeps

track of the value of the last path square consumed by the agent. The only difference between computeFSMPCFitness and computeFSMFitness involves the fitness calculation in step 1. The agent increases its fitness value by one only if the value of the current path square consumed by the agent is greater than the value of the previous path square consumed by the agent. Section 4.5 in chapter 4 provides a detailed description of the methods to compute an agent's fitness value. The next section details the methods used by the genetic algorithm to select and reproduce the agents for each generation of the signature recognition genetic algorithm.

2.4 Reproduction of the Next Generation

The signature recognition genetic algorithm has a population of 16,000 agents with each iteration of the genetic algorithm producing a new generation of 16,000 agents. Two parent agents, chosen from the current population, breed an offspring agent of the new generation. The roulette wheel selection method enables the genetic algorithm to randomly choose an offspring's parents. Subsection 2.4.1 defines and illustrates the use of roulette wheel selection. Once the parents are chosen, the crossover operator creates the offspring from portions of each parent. Subsection 2.4.2 details the operation of the crossover operator. Finally, the application of mutation to each offspring introduces variety into the agent's genome as described in subsection 2.4.3.

2.4.1 The Roulette Wheel Selection Method

The genetic algorithm roulette wheel is a genetic algorithm selection method based on a gambler's roulette wheel to randomly choose an individual from the

population. Thirty-eight equal sized sections divide the gambler's roulette wheel with 37 sections numbered 0 through 36 and the 38th section numbered 00. The croupier spins the wheel in one direction and throws the marble in the opposite direction. Eventually the wheel and the marble stop moving with the marble coming to rest in one of the sections. This concept translates to a selection method in a genetic algorithm.

Each agent in the population occupies one section of the genetic algorithm roulette wheel where the agent's fitness value determines the size of the section. The genetic algorithm equivalent of the roulette wheel marble, generated for each selection of an individual from the population, is a randomly chosen integer between 0 and the sum total of the population's fitness values. Starting with the population's first individual, simulation of the motion of the wheel and marble executes the following two steps. Step 1 - is the marble's value less than the fitness value of the current individual? Step 2 - if the answer to the question in step 1 is yes then choose the current individual otherwise subtract the current individual's fitness value from the marble's value and repeat both steps for the next member of the population. Members of the population with higher fitness values have a higher chance of selection.

Individual	Fitness Value
agent 1	19
agent 2	25
agent 3	7
agent 4	14
agent 5	2
Sum total of the population's fitness values: 67	

Figure 2.3 - An example population.

For example, consider the small example population in figure 2.3 listing each agent's fitness value. Assume 54 is the roulette wheel marble; a randomly chosen value from the range of 0 to 67, the sum total of the population's fitness values from figure 2.3. Begin with agent 1 in figure 2.3. According to step 1, the marble's value 54 is not less than agent 1's fitness value 19. In this case, step 2 instructs the roulette wheel to subtract agent 1's fitness value 19 from the marble's value 54 to make the marble's value 35 and repeat steps 1 and 2 for the next member of the population, agent 2. According to step 1, the marble's value 35 is not less than agent 2's fitness value 25. In this case, step 2 instructs the roulette wheel to subtract agent 2's fitness value 25 from the marble's value 35 to make the marble's value 10 and repeat steps 1 and 2 for the next member of the population, agent 3. According to step 1, the marble's value 10 is not less than agent 3's fitness value 7. In this case, step 2 instructs the roulette wheel to subtract agent 3's fitness value 7 from the marble's value 10 to make the marble's value 3 and repeat steps 1 and 2 for the next member of the population, agent 4. According to step 1, the marble's value 3 is less than agent 4's fitness value 14. In this case, step 2 instructs the roulette wheel to choose agent 4 as the individual selected. The next subsection describes the means to create the crossover points for the genetic algorithm agents as well as an example of the application of crossover to create an offspring agent from two parent agents.

2.4.2 The Crossover Operator

Definition

Crossover is a genetic algorithm operator that creates an offspring from two parents.

Definition

Crossover point is a position within a segment of an individual's chromosome that divides the segment of the chromosome into two pieces.

The signature recognition program creates the crossover points for the agents before the program uses the genetic algorithm. Subsection 2.4.2.1 provides a brief explanation of the crossover point creation process. Subsection 2.4.2.2 describes an application of crossover for a single crossover point.

2.4.2.1 Creation of the Crossover Points

The agent's finite state machine is a (rectangular) two-dimensional array of 32 rows and 3 columns. For the signature recognition genetic algorithm, a crossover point is a rectangular sub-area of the agent. All agents in the population use the same set of crossover points. The intersection of the crossover point areas is zero and the union of the crossover point areas is equal to the total area of the agent. The algorithm to create the set of crossover points essentially divvies up the agent's area, through random choices, into the crossover points. A queue stores areas to process, initialized by inserting the agent's area into the queue. The algorithm removes an area A from the front of the queue having length LA and width WA. The new crossover point C is a rectangular piece of area A. The length LC of C is a randomly chosen integer from the range of 1 to LA and the width WC of C is a randomly chosen integer from the range of 1 to WA. The algorithm places the crossover point in a randomly chosen corner within area A and places the remaining area of A into the queue for further processing. The shape of the remaining area depends on the size of the crossover point. There is no remaining area if LC = LA and WC = WA. If LC = LA or WC = WA, the remaining area

is a single rectangular piece with the piece placed into the queue. If LC < LA and WC < WA, the remaining area is an L shaped piece. Cut the L shaped remaining area into two rectangular pieces placing each piece into the queue. The algorithm processes the areas from the queue until the queue is empty. Section 4.6 in chapter 4 provides a detailed description of the methods to create the crossover points. The next subsection describes an application of crossover for a single crossover point.

2.4.2.2 The Application of Crossover

After selection of both parents, crossover creates an offspring by mixing values from each parent. Crossover utilizes a crossover rate and the queue of crossover points. The crossover rate is a real number between 0 and 1. Process each crossover point in the queue as follows. Randomly generate a real number x between 0 and 1. If x is greater than the crossover rate then copy every "output/next state" value from within the crossover point area of parent 1 to the corresponding place within the same crossover point area in the offspring; otherwise, copy every "output/next state" value from within the crossover point area of parent 2 to the corresponding place within the same crossover point area in the offspring. Figure 2.4 presents the application of a crossover point to an offspring with a crossover rate of 0.07 and a randomly chosen value of 0.5 for x. The crossover area is the area surrounded by the bold dashed line. The next subsection describes the application of mutation to an individual.

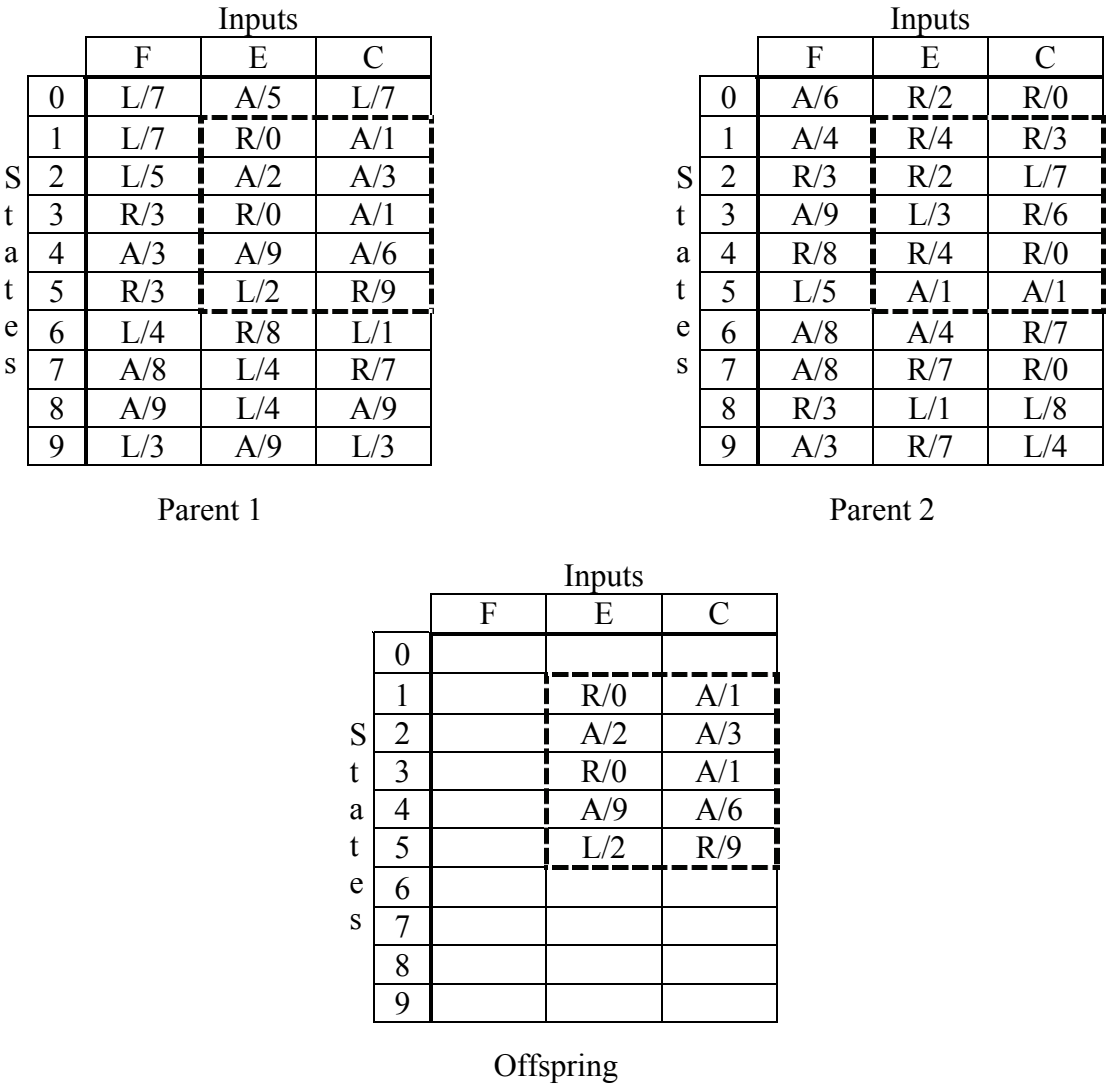


Figure 2.4 - The application of a crossover point.

2.4.3 The Mutation Operator

Definition

Mutation is a genetic algorithm operator that changes one or more bits of an individual by random means.

Application of mutation to the offspring happens after processing of all of the crossover points. The genetic algorithm has a mutation rate that is a real number between

0 and 1. For each output in the offspring, randomly generate a real number y between 0 and 1. If y is less than or equal to the mutation rate then replace the output with a randomly chosen valid output value. For each next state in the offspring, randomly generate a real number z between 0 and 1. If z is less than or equal to the mutation rate then replace the next state with a randomly chosen valid next state value. The next section describes the signature recognition genetic algorithm.

2.5 A Genetic Algorithm for the Signature Recognition Problem

The signature recognition genetic algorithm executed within the signature recognition program incorporates the structures and operators described in the previous sections. The genetic algorithm for this problem evolves a population of agents, searching for an agent or agents that can traverse the entire length of the current path in the grid. Figure 2.5 displays the flowchart of the signature recognition genetic algorithm with section 4.7 in chapter 4 providing a detailed description of the signature recognition genetic algorithm method.

The signature recognition tuning program tunes the crossover and mutation rates of the genetic algorithm to optimum values before beginning the signature recognition program experiments. Chosen from one of the pairings of a crossover rate and a mutation rate: [5%, 6%, 7%, 8%, 9%, 10%] × [1%, 2%, 3%] are the crossover and mutation rates used in the signature recognition program experiments. Initial empirical experiments tested a wide range of crossover and mutation rates to narrow the rates down to the set of best performing rates, the rates listed in the previous sentence.

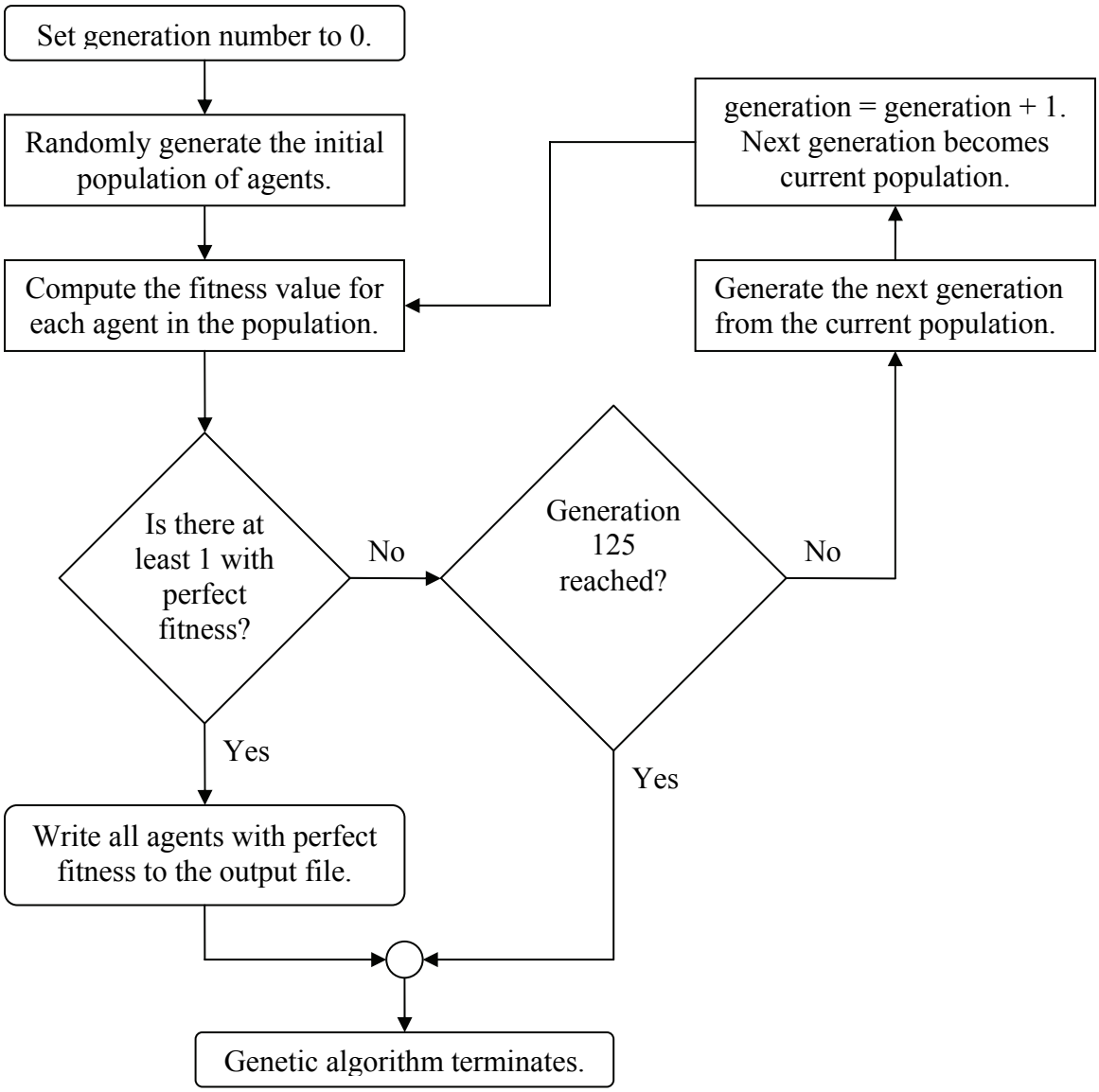


Figure 2.5 - Flowchart of the genetic algorithm.

The signature recognition genetic algorithm tuning program uses five paths of various lengths (562, 1224, 763, 1126 and 503) executing the signature recognition genetic algorithm once for each path and each crossover and mutation rate pair. The signature recognition genetic algorithm writes to the output file each agent with perfect fitness along with the path, run and generation number that produced the perfectly fit agent. An examination of the output file produced by the signature recognition tuning

program reveals the crossover and mutation pair that generated agents with perfect fitness in the least number of generations. It is this pair of rates that the signature recognition program uses in the main signature recognition program experiments.

One instance of the tuning program ran on a computer, called Intrepid, having a Pentium 266 megahertz processor with 96 megabytes of random access memory running Microsoft Windows 2000. Another instance of the tuning program ran on a Sun UltraSPARC 10 computer, called Eniac, having a 300 megahertz UltraSPARC IIi processor with 128 megabytes of random access memory running Solaris 8/SunOS 5.8. In figure 2.7, there is one table for each of the five paths processed by the tuning program on Intrepid. Each entry in a table corresponds to a particular crossover and mutation rate pair and contains the number of the last generation processed by the genetic algorithm upon termination as written to the output file TuneGA.txt. The sixth table in figure 2.7 is the average of the first five tables. Figure 2.8 provides the same data as figure 2.7 but for the execution of the tuning program on Eniac. The table in figure 2.6 provides the average of the sixth tables in figures 2.7 and 2.8. The crossover rate of 7% and mutation rate of 3% was the pair in which the genetic algorithm terminated at the earliest generation. Thus, these are the rates used during the execution of the signature recognition program described in the next section.

Average generation over both Intrepid and Eniac.

	Crossover Rate					
Mutation Rate	5%	6%	7%	8%	9%	10%
1%	7.0	6.6	8.0	6.6	6.2	6.6
2%	5.2	6.0	4.4	6.4	5.4	7.0
3%	5.0	5.2	4.2	4.8	5.0	4.4

Figure 2.6 - The average of the generation the genetic algorithm terminated on Intrepid and Eniac for each crossover and mutation rate pair.

Path 1 – Length 562

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	2	2	8	2	4	2
2%	6	6	6	4	4	2
3%	4	4	2	6	4	4

Path 2 – Length 1224

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	12	22	20	18	10	14
2%	12	14	10	20	12	20
3%	8	8	10	10	10	10

Path 3 – Length 763

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	2	8	4	2	2	2
2%	2	2	2	2	2	2
3%	2	2	2	2	2	2

Path 4 – Length 1126

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	2	2	2	2	2	2
2%	2	2	2	2	2	2
3%	2	2	2	2	2	2

Path 5 – Length 503

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	20	10	12	14	12	14
2%	14	12	8	6	12	6
3%	10	10	8	8	10	8

Average over all five paths.

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	7.6	8.8	9.2	7.6	6.0	6.8
2%	7.2	7.2	5.6	6.8	6.4	6.4
3%	5.2	5.2	4.8	5.6	5.6	5.2

Figure 2.7 - The generation the genetic algorithm terminated for each path's crossover and mutation rate pairs as executed on a Windows 2000 computer.

Path 1 – Length 562

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	6	2	4	6	4	8
2%	4	2	2	4	4	6
3%	2	4	2	2	4	2

Path 2 – Length 1224

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	16	8	16	10	16	10
2%	2	12	4	18	6	20
3%	6	12	6	8	6	6

Path 3 – Length 763

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	2	2	2	2	2	2
2%	2	2	2	2	2	2
3%	2	2	2	2	2	2

Path 4 – Length 1126

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	0	0	0	0	0	0
2%	0	0	0	0	0	0
3%	0	0	0	0	0	0

Path 5 – Length 503

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	8	10	12	10	10	12
2%	8	8	8	6	10	10
3%	14	8	8	8	10	8

Average over all five paths.

Mutation Rate	Crossover Rate 5%	6%	7%	8%	9%	10%
1%	6.4	4.4	6.8	5.6	6.4	6.4
2%	3.2	4.8	3.2	6.0	4.4	7.6
3%	4.8	5.2	3.6	4.0	4.4	3.6

Figure 2.8 - The generation the genetic algorithm terminated for each path's crossover and mutation rate pairs as executed on a Sun UltraSparc 10 computer.

2.6 The Signature Recognition Program

The signature recognition genetic algorithm is a method that executes inside of the signature recognition program. Hence, the signature recognition program is a driver program for the signature recognition genetic algorithm. Figure 2.9 displays the flowchart of the signature recognition program with section 4.8 in chapter 4 providing a detailed description of the signature recognition program's main method.

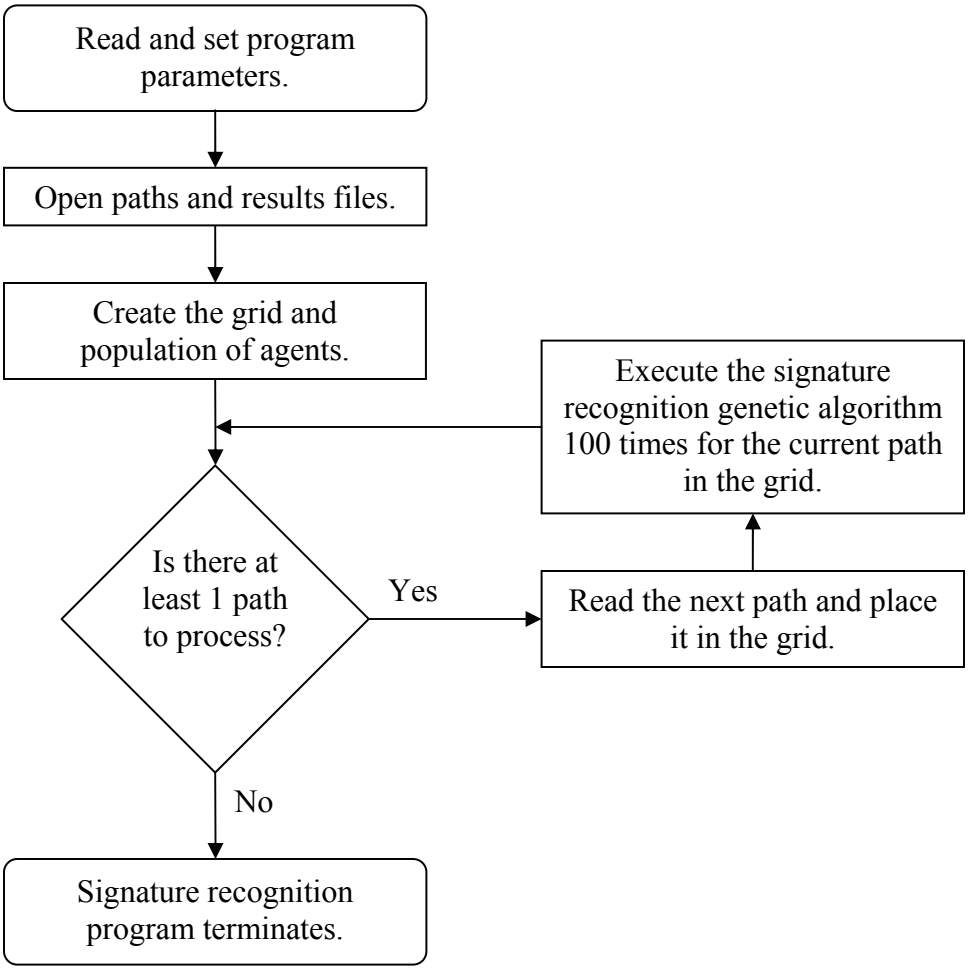


Figure 2.9 - Flowchart of the signature recognition program.

For each experiment, 100 instances of the signature recognition program execute in parallel. Each instance processes the 100 paths from one of the paths files. Therefore, an experiment processes a total of 10,000 paths from one of the data sets. For each path, the signature recognition genetic algorithm executes 100 times.

Some instances of the signature recognition program ran on computers with Intel processors running a version of the Microsoft Windows operating system. Subsection 2.6.1 provides the specifications of the Windows computers as well as the means by which the signature recognition program ran on these computers. Other instances of the signature recognition program ran on a computational cluster using computers with Intel processors running the Red Hat Linux operating system. Subsection 2.6.2 provides the specifications of the computational cluster computers as well as the means by which the signature recognition program ran on these computers. Finally, subsection 2.6.3 provides a brief description of each of the five signature recognition program experiments.

2.6.1 The Program Execution under Microsoft Windows

For each of the five experiments, various instances of the signature recognition program ran on Windows computers. There were five groups of Windows computers that ran the signature recognition program with a description of each group's computer hardware provided here. The 28 personal computers in the first group had a Pentium III class Celeron 766 megahertz processor with 256 megabytes of random access memory running the Microsoft Windows XP operating system. The 30 personal computers in the second group had a Pentium 4 class Celeron 1 gigahertz processor with 256 megabytes of random access memory running the Microsoft Windows 2000 operating system. The 30

personal computers in the third group had a Pentium III 933 megahertz processor with 256 megabytes of random access memory running the Microsoft Windows XP operating system. The 30 personal computers in the fourth group had a Pentium 4, 3 gigahertz processor with 512 megabytes of random access memory running the Microsoft Windows XP operating system. The 20 personal computers in the fifth group had a Pentium D 2.8 gigahertz dual core processor with 2 gigabytes of random access memory running the Microsoft Windows XP operating system. A Microsoft Visual C++ compiler provides an executable version of the signature recognition program and a batch file installs an instance of the signature recognition program for execution on a Windows computer. Section 4.9 in chapter 4 provides a detailed description of the batch file and method to execute the signature recognition program on a Windows computer. The next subsection provides the specifications of the computational cluster computers as well as the means by which the signature recognition program ran on these computers.

2.6.2 Programming in a Distributed Computing Environment

For each of the five experiments, various instances of the signature recognition program ran on the computational cluster. There were two groups of computers that ran the signature recognition program with a description of each group's computer hardware provided here. The 30 computers in the first cluster group are Dell 1550 computers having dual Pentium III 933 megahertz processors with 1 gigabyte of random access memory running the Red Hat Linux operating system. The 30 computers in the second cluster group have a Pentium 4, 2.6 gigahertz processor with 1 gigabyte of random access memory running the Red Hat Linux operating system. A Portland Group C++ compiler

provides an executable version of the signature recognition program for execution on the cluster's computers.

All 60 of the cluster's computers are accessible indirectly via a user account on the cluster's host computer running the Red Hat Linux operating system. The Sun Grid Engine (SGE) is the application that schedules a program for execution on one of the cluster's computers. The user's programs executing on the cluster computers use and share the disk storage of the user account's home directory. The execution of a program on the cluster takes the form of a shell script defined batch job. Each submission of a batch job to the SGE begins execution of another instance of the signature recognition program. Section 4.10 in chapter 4 provides a detailed description of the batch file and method to execute the signature recognition program on the computational cluster. The next subsection provides a brief description of the five experiments of the signature recognition program.

2.6.3 Testing the Signature Recognition Program

The first experiment of the signature recognition program processes the 10,000 non-crossover paths using the non-contiguous consumption fitness function computeFSMFitness described in section 2.3. The second experiment of the signature recognition program also processes the 10,000 non-crossover paths using the non-contiguous consumption fitness function computeFSMFitness described in section 2.3. The difference between the first and second experiments is the two part processing done to each agent before computing the agent's fitness value.

The first part of processing an agent determines the states in the agent's finite state table that are reachable from the start state 0. The start state of all agents is always state 0. A Boolean array called *reachable* keeps track of which states are reachable (true) and which states are not reachable (false) from the start state 0. The number of elements in *reachable* is equal to the number of states in an agent with each element initialized to false. An integer queue called *processing* contains states whose next state values need processing. The queue *processing* is initially empty. The algorithm to search for an agent's reachable states begins by placing the start state 0 into the queue *processing*. The algorithm executes the following steps so long as the queue *processing* is not empty. Step 1: remove the state at the head of the queue making it the current state. Step 2: mark the current state as reachable by setting the current state's corresponding element in the array *reachable* to true. Step 3: for each finite state machine input value for the current state, retrieve the next state number. If the next state is not set as reachable and is not in the queue *processing*, place the next state at the end of the queue *processing*. Step 4: if the queue *processing* is not empty, return to step 1; otherwise, the algorithm terminates. Once the algorithm terminates, all the states in the array *reachable* whose value is true are reachable from the start state 0. For the agent in figure 2.10, the states 0, 1, 2, 3, 5 and 6 are reachable from the start state 0, whereas states 4 and 7 are not reachable from the start state 0.

For the second experiment, the second part of processing the agent is compaction. Compaction groups all of the reachable states together at the top of the agent's finite state table and groups all of the non-reachable states at the bottom of the agent's finite state table. For the agent in figure 2.10, the reachable states 0, 1, 2, and 3 remain in place.

Non-reachable state 4 swaps positions with reachable state 5. Then, all next state 4 values in the agent's finite state table become next state 5 values and vice versa. The now non-reachable state 5 swaps positions with reachable state 6. Then, all next state 5 values become next state 6 values and vice versa. The non-reachable state 7 remains in place. Now, states 0 through 5 are reachable from the start state 0 and states 6 and 7 are not reachable from the start state 0. Figure 2.11 illustrates the agent from figure 2.10 after compaction.

		Inputs		
		S	E	C
States	0	L/2	A/0	L/1
	1	L/6	R/5	A/1
	2	L/5	A/6	A/2
	3	R/2	R/0	A/5
	4	A/2	A/5	A/2
	5	R/1	L/5	R/2
	6	L/6	R/3	L/2
	7	L/7	A/1	R/5

Figure 2.10 - An example of an agent.

		Inputs		
		S	E	C
States	0	L/2	A/0	L/1
	1	L/5	R/4	A/1
	2	L/4	A/5	A/2
	3	R/2	R/0	A/4
	4	R/1	L/4	R/2
	5	L/5	R/3	L/2
	6	A/2	A/4	A/2
	7	L/7	A/1	R/4

Figure 2.11 - An agent after compaction.

The third experiment of the signature recognition program also processes the 10,000 non-crossover paths using the non-contiguous consumption fitness function computeFSMFitness described in section 2.3. The difference between the second and third experiments is the second part of the two part processing done to each agent before computing the agent's fitness value.

For the third experiment, the second part of processing the agent is spreading which spaces the reachable states and non-reachable states evenly throughout the agent's finite state table. Actually, spreading evenly spaces only the states in the group with the least number of states. For the agent in figure 2.10, the number of reachable states is 6 and the number of non-reachable states is 2 making the non-reachable states evenly spaced throughout the agent's finite state table. Divide the total number of states in the agent by the number of states in the smaller sized group to get the value i. Therefore, every i^{th} state becomes a state from the smaller sized group unless the i^{th} state is already a state from this group. For the agent in figure 2.10, i = 4 = 8 / 2. Therefore, every 4^{th} state becomes a non-reachable state, states 3 and 7. State 3 is a reachable state and state 4 is a non-reachable state. Reachable state 3 swaps positions with non-reachable state 4. Then, all next state 3 values in the agent's finite state table become next state 4 values and vice versa. State 7 is already a non-reachable state, ending the required number of moves. Figure 2.12 illustrates the agent from figure 2.10 after spreading.

The fourth experiment of the signature recognition program processes the 10,000 crossover paths using the non-contiguous consumption fitness function computeFSMFitness as described in section 2.3. The fifth experiment of the signature recognition program processes the 10,000 crossover paths using the partial contiguous

consumption fitness function computeFSMPCFitness as described in section 2.3. Chapter 6 contains the description and discussion of the results of each of the experiments of the signature recognition program. The next section describes the program to provide data about the landscape of the search space of the signature recognition problem.

		Inputs		
		S	E	C
	0	L/2	A/0	L/1
S	1	L/6	R/5	A/1
t	2	L/5	A/6	A/2
a	3	A/2	A/5	A/2
t	4	R/2	R/0	A/5
e	5	R/1	L/5	R/2
s	6	L/6	R/4	L/2
	7	L/7	A/1	R/5

Figure 2.12 - An agent after spreading.

2.7 The Landscape of the Signature Recognition Program Search Space

A view of the landscape of the signature recognition program search space helps in the analysis of the results of the program's experiments. 100 instances of the landscape program execute in parallel. Each instance processes the 100 paths from one of the paths files of the non-crossover paths data set. Therefore, all instances of the landscape program process a total of 10,000 paths. Section 4.11 in chapter 4 provides a detailed description of the landscape program's main method. The landscape program opens the paths file reading all 100 paths into the program, opens the output file to store the results, and creates an instance of an agent. The landscape program performs the

following 3 steps 1,000,000 times. Step 1: randomly choose a valid output value and a valid next state value for each entry in the agent's finite state table. Step 2: compute the number of reachable states for the agent and write the number to the output file. Step 3: compute the agent's fitness value for each of the 100 paths, writing each fitness value to the output file. The landscape program writes the number of reachable states and all 100 fitness values for the agent to a single line. The output file will contain 1,000,000 lines, one for each randomly created agent. Chapter 6 contains the description and discussion of the results of the landscape program. The next chapter describes the second problem endeavored in this dissertation.

Chapter 3 Moment Based Optical Character Recognition

Moments are statistical region-based shape descriptors for objects in an image. Moments are region-based because they utilize both boundary and internal pixels to calculate their value for the object. The basic moment (m_{pq}) equation, figure 3.1, calculates its value from the coordinates and the color value of the pixels within the area bounding the object. The function f(x, y) represents the color value in the moment equation of figure 3.1.

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y) \text{ where p, q = 0, 1, 2, …}$$

Figure 3.1 - The basic moment equation.

$$m_{00} = \sum_x \sum_y f(x, y)$$

Figure 3.2 - The moment equation for the area of the object.

$$m_{10} = \sum_x \sum_y xf(x, y)$$
$$m_{01} = \sum_x \sum_y yf(x, y)$$

$$\bar{x} = m_{10} / m_{00}$$
$$\bar{y} = m_{01} / m_{00}$$

Figure 3.3 - The center of mass moment equations.

The area of the object is the moment equation m_{00}, figure 3.2. The center of mass moment equations, figure 3.3, base themselves on the area moment m_{00} and the moment's m_{10} and m_{01}. These simple moments have the problem of not being translation, rotation and scaling invariant. The central moment equation, figure 3.4, is translation invariant through the use of the center of mass moments.

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \text{ where p, q = 0, 1, 2, …}$$

Figure 3.4 - The central moment equation.

The central moment equation with certain values for p and q represent various properties of the object. The first order central moment μ_{00} is just the mean of the object, its area. The second order central moment's μ_{20} and μ_{02} define the object's variance, the spread of points around the object's center of mass. A better second order central moment is μ_{11}, covariance, because the spread of points may not be evenly aligned with the axis. The third order central moment's μ_{30}, μ_{03}, μ_{21} and μ_{12} measure the skew of the object, the spread of points around the object's mean. The fourth order central moment's μ_{40}, μ_{04}, μ_{31}, μ_{31} and μ_{22} measure the object's kurtosis distribution, the shape (flat or sharp) of the peak of the point's distribution. The complex moment equations of Hu and Flusser-Suk employ these central moments to generate moment values that are invariant to translation, rotation and scaling which are useful for optical character recognition. The next section begins the discussion of the optical character recognition system employing the Hu and Flusser-Suk invariant moments.

3.1 An Introduction to the Optical Character Recognition System

An ordinary flatbed scanner scans an 8.5 by 11 inch page of printed text storing the resulting image in a file using the TIFF file format. To the naked eye, the background pixels in the image are white and the pixels making up the potential characters are black but the TIFF image is a 24-bit color image. The optical character recognition (OCR) program requires a binary (black/white) color high-contrast version of the original image. The threshold command in Adobe Photoshop is a quick way to apply image thresholding

to the original image file to obtain the binary color image. A value between 0 and 255 specifies the level of thresholding. Each pixel that is less than the threshold level becomes a black pixel and each pixel that is greater than the threshold level becomes a white pixel. Experimentation of various threshold level values over several images of various fonts and font sizes determined that a threshold level of 161 provided the best contrast.

This dissertation's moment based solution to the optical character recognition problem employs a two stage process. The first stage executes the character database creation (CDC) program to create a database of all printable characters with each character presented in several different combinations of font name and font size. The second stage performs the moment based optical character recognition through execution of the optical character recognition (OCR) program on pages of text.

Here is a brief description of the major phases of the OCR program. The first phase of the OCR program finds all of the connected components in the image. A connected component is a set of black pixels in the image in which each black pixel in the component is a neighbor to at least one other pixel in the component. None of the pixels in one component are neighbors of pixels in any other component. Once detection of all connected components completes, each component computes its structures and statistical values. The second phase uses a genetic algorithm to determine the number of lines of text in the image. The third phase uses the number of lines to arrange all of the image's components by line. Neighboring components on a line within a certain distance threshold of each other merge into a single component. The fourth phase computes the 7

Hu and 10 Flusser-Suk moment values for each component. The fifth and final phase of the OCR program performs the character recognition for each component.

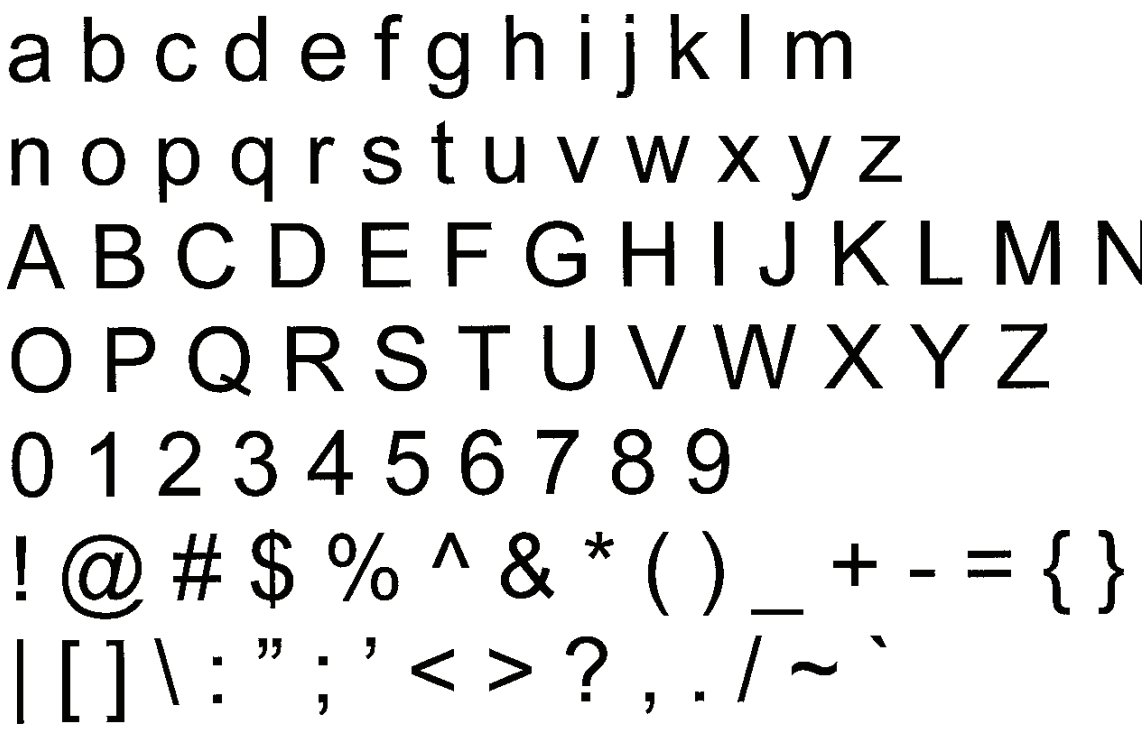


Figure 3.5 - Set of all printable characters with font name and size of Arial 36 point.

The CDC program uses the same first four phases as the OCR program. Hence, characters in images are the characters added into the database. A single image provides the CDC program with all of the printable characters on the U.S. English keyboard for a particular font name and font size combination. Figure 3.5 is an example of the image for the characters of the font name and font size combination Arial 36 point. The fifth and final phase of the CDC program stores all of the image's characters, of a particular font name and font size combination, into the database. The CDC program processes more than one font name and font size combination by repeatedly carrying out its five phases on each font name and font size combination image supplied as input.

The discussion of the optical character recognition system begins in the next section with the first major phase in both the OCR and CDC programs, recognition of the connected components in the image. After recognition of the connected components, section 3.3 discusses the computation of basic statistical values for each connected component. Section 3.4 details the next major phase in the OCR and CDC programs execution of the line recognition genetic algorithm. The description in section 3.5 details the organization, by line, of all of the connected components in the image after execution of the line recognition genetic algorithm. Section 3.6 details the point at which the two programs, OCR and CDC, diverge from common code, to code that is unique to each program. Finally, section 3.7 provides a brief description of the means for verification of the moment computation code, described in section 3.6.

3.2 Recognition of Connected Components

Both, the OCR and CDC programs begin by finding the connected components in the binary color image. Recall from section 3.1, a connected component is a set of black pixels in the image in which each black pixel in the component is a neighbor to at least one other pixel in the component. None of the pixels in one component are neighbors of pixels in any other component.

Assisting in the search for the components are two data structures: a one-dimensional array of linked lists called complists, and a two-dimensional array of integers, called compgrid. Each linked list in complists stores the coordinates of every pixel of a unique connected component. Initially, all the linked lists are empty. The size of the array complists is set to the maximum possible number of components that can

exist in an image which is possible if every component is smallest in size, made up of a single pixel. Arraignment of these single pixel components in the image is as follows. Every odd numbered row contains components. Every even numbered row contains background (white) colored pixels in order to separate components in neighboring rows. The integer variable, rows, contains the number of rows in the image. Hence, the expression ((int) (rows / 2.0 + 0.5)) yields the number of rows in the image having components. Within an odd numbered row, every odd numbered pixel is a component and every even numbered pixel is white in order to separate neighboring components within the row. The integer variable, cols, contains the number of columns in the image. Hence, the expression ((int) (cols / 2.0 + 0.5)) yields the number of components within an odd numbered row. Therefore, the size of complists, called numlists, computes as [((int) (rows / 2.0 + 0.5)) * ((int) (cols / 2.0 + 0.5))].

The array compgrid keeps track of the component assigned to each pixel in the image. The size of compgrid is (rows x cols) having the same number of rows and columns as the image. Each index in the array complists denotes a particular component. The indices of complists range from 0 to (numlists-1). The value of -1, called NO_COMP, specifies that a pixel does not belong to any component. The initial value of all elements of array compgrid is NO_COMP.

The connected component recognition algorithm examines each pixel in the image one at a time starting with the pixel at coordinates (0, 0), row 0 and column 0 in the upper left hand corner of the image. Assume that the current pixel is the pixel at coordinates (i, j). If the color of the current pixel is white, the value of the element in compgrid corresponding to the current pixel remains as NO_COMP. If the color of the

current pixel is black, the coordinates of the current pixel pigeonholes into the appropriate component's linked list in complists. For the current pixel, the only neighboring pixels processed already are the four pixels at coordinates (i-1, j-1), (i-1, j), (i-1, j+1) and (i, j-1), see figure 3.6. The algorithm retrieves from compgrid the component number for each of these four pixels discarding the NO_COMP values, white pixel neighbors. The current pixel belongs to one of the remaining components. The next actions taken depend on the number of remaining unique component numbers of these four neighbors.

(i-1, j-1)	(i, j-1)	(i+1, j-1)
(i-1, j)	(i, j)	

Figure 3.6 - The coordinates of the processed neighbors of the pixel (i, j).

If the number of remaining unique component numbers is zero, the current pixel (i, j) forms a new component. The algorithm inserts the coordinates of the current pixel (i, j) into the first empty linked list in complists thereby forming the new component. Assign the current pixel's new component number to the current pixel's corresponding element in compgrid.

If the number of remaining unique component numbers is one, the current pixel is a part of that one unique component. The algorithm inserts the coordinates of the current pixel (i, j) into the linked list of that component. Assign the current pixel's new component number to the current pixel's corresponding element in compgrid.

If the number of remaining unique component numbers is two or more, all of the unique components merge into the component with the smallest unique component number leaving the linked lists for the remaining unique components empty, for reuse,

except the component with the smallest unique component number. The current pixel is a part of the merged component. The algorithm inserts the coordinates of the current pixel (i, j) into the linked list of the merged component. Assign the current pixel's new component number to the current pixel's corresponding element in compgrid. For each pixel in the merged component, ensure the pixel's corresponding element in compgrid is equal to the merged component's number.

Once finished with the current pixel (i, j), the algorithm moves on to the pixel (i+1, j) to the right of the current pixel. If the current pixel is the last pixel in the row, the algorithm moves to the first pixel in the next row. Once processing of all of the pixels in the image completes, each non-empty linked list of pixels in complists represents a unique connected component in the image. After recognition of the connected components, the next section discusses the computation of basic statistical values for each connected component.

3.3 Statistics Required for the Connected Components

The OCR and CDC programs utilize a Component class to store the values and statistics of a connected component. A one-dimensional array of Component, called *components*, holds the entire image's connected components. Each element of *components* is a unique instance of the Component class and receives a unique linked list from complists thereby storing one of the image's connected components. Each connected component computes its statistics with each value and structure stored in the connected component's instance in the array *components*. The first statistic is the total number of pixels in the connected component. The second statistic is the coordinates of

the centroid pixel of the connected component computed as seen in figure 3.7. x_{min} and y_{min} are the smallest x and y coordinates, respectively, of all of the pixels in the connected component. x_{max} and y_{max} are the largest x and y coordinates, respectively, of all of the pixels in the connected component. These four values form the coordinates of the smallest axis aligned rectangle that encompasses the connected component. Figure 3.8 illustrates the smallest axis aligned rectangle surrounding a connected component.

$$\left(\frac{\sum_{i=1}^{j} x_i}{j}, \frac{\sum_{i=1}^{j} y_i}{j} \right)$$

Figure 3.7 - Coordinates of the centroid pixel of a component, where j is the number of pixels in the component.

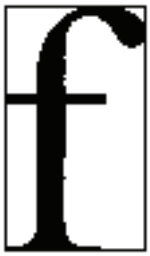


Figure 3.8 - The smallest axis aligned rectangle surrounding the character f.

There are three structures computed for each connected component: the pixels of the connected component's outer border, the pixels of the connected component's convex hull, and the connected component's minimum area rectangle, the rectangle with the smallest area that surrounds the connected component. Figure 3.9 illustrates the minimum area rectangle surrounding a connected component. The algorithm to compute the minimum area rectangle is from a paper written by David Eberly (2003) and requires the convex hull of the connected component. The algorithm to compute the convex hull

requires the pixels of the connected component's outer border. The next subsection describes the outer border algorithm with the convex hull algorithm described in subsection 3.3.2.

Figure 3.9 - The minimum area rectangle surrounding the character f.

3.3.1 Computing the Outer Border of a Connected Component

The height, called h, of the connected component calculates as $y_{max} - y_{min} + 1$ and the width, called w, calculates as $x_{max} - x_{min} + 1$. Create an image, called *canvas*, whose height is h+2 and width is w+2. Initialize all the pixels in *canvas* to the color white. Mapping each pixel of the connected component to its corresponding pixel in *canvas* draws the connected component inside *canvas*. Assume the coordinates of a connected component's pixel is (x, y), then the coordinates of its corresponding pixel in *canvas* is ($x - x_{min} + 1$, $y - y_{min} + 1$). This places the connected component within rows 1 through h and columns 1 through w inside *canvas* providing a border of white pixels surrounding the connected component to make outer border detection easier to perform. Figure 3.10 illustrates a connected component with width 8 and height 8 drawn in the image canvas with width 10 and height 10.

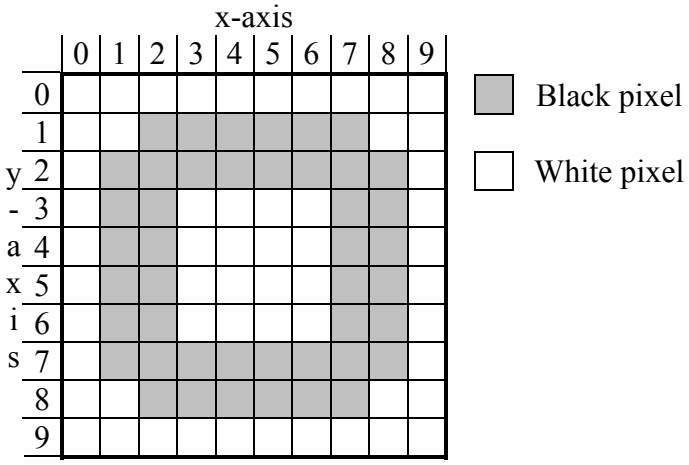


Figure 3.10 - A black and white 10 by 10 pixel image.

After the initialization, all of the white pixels on the outside of the connected component in canvas change to black via the use of a pixel stack and the following procedure. Initially, the stack is empty. The procedure starts by pushing the pixel at coordinates (0, 0) onto the stack. Step 1 – remove the pixel at the top of the stack, whose coordinates are (x, y), making it the current pixel. Change the current pixel to black. Step 2 - for the following four pixels, place the white pixels onto the stack: the pixel above (x, y-1), below (x, y+1), to the left (x-1, y), and to the right (x+1, y) of the current pixel. Repeat steps 1 and 2 until the stack is empty. At this point, all of the black pixels in *canvas* are either a pixel of the connected component or a pixel on the outside of the connected component. Any white pixels in *canvas* are inside an area that is completely surrounded by the connected component. Figure 3.11 displays *canvas* after application of the procedure from this paragraph to *canvas* from figure 3.10.

Next, reverse the colors of the pixels in *canvas*. For each pixel in *canvas*, change a white pixel to black and a black pixel to white. At this point, all the white pixels in *canvas* are either a pixel of the connected component or a pixel on the outside of the

connected component. Any black pixels in *canvas* are in an area that is completely surrounded by the connected component. Figure 3.12 displays *canvas* after reversing the colors of *canvas* from figure 3.11.

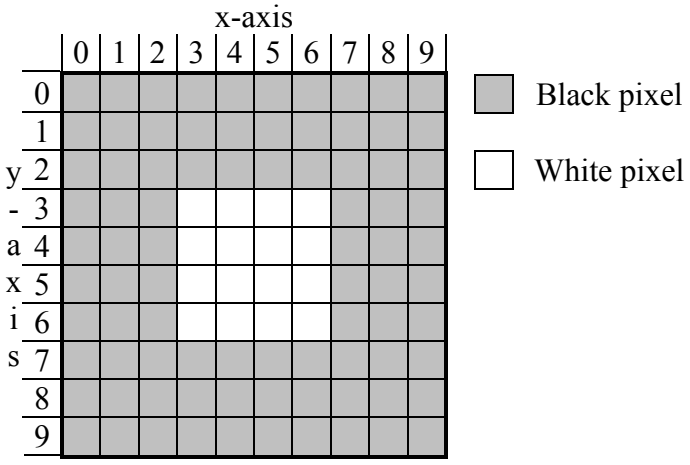


Figure 3.11 - A black and white 10 by 10 pixel image.

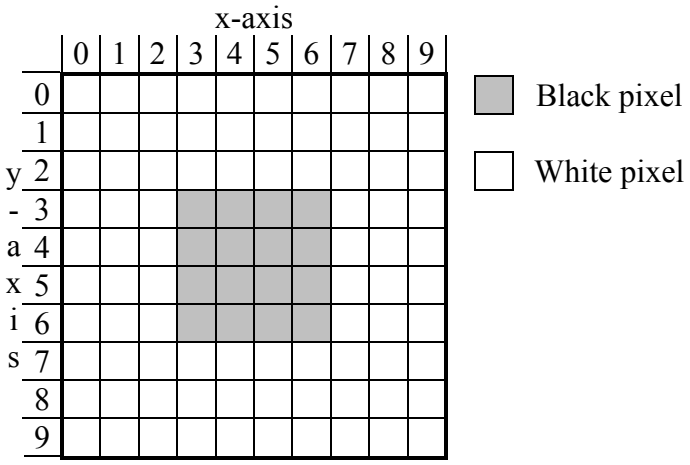


Figure 3.12 - A black and white 10 by 10 pixel image.

Draw the connected component in *canvas* again. At this point, all the white pixels in *canvas* are pixels on the outside of the connected component. Figure 3.13 displays *canvas* after drawing the connected component in the *canvas* from figure 3.12. For each pixel in the image, if the pixel is black, determine if any of the eight pixels [(x-1, y-1), (x, y-1), (x+1, y-1), (x-1, y), (x+1, y), (x-1, y+1), (x, y+1) and (x+1, y+1)] surrounding the black pixel (x, y) is white. If this is the case then the black pixel is a border pixel. Insert the coordinates of the border pixels into the border linked list kept in the connected component's instance in the array *components*. Figure 3.14 illustrates the outer border from the *canvas* of figure 3.13.

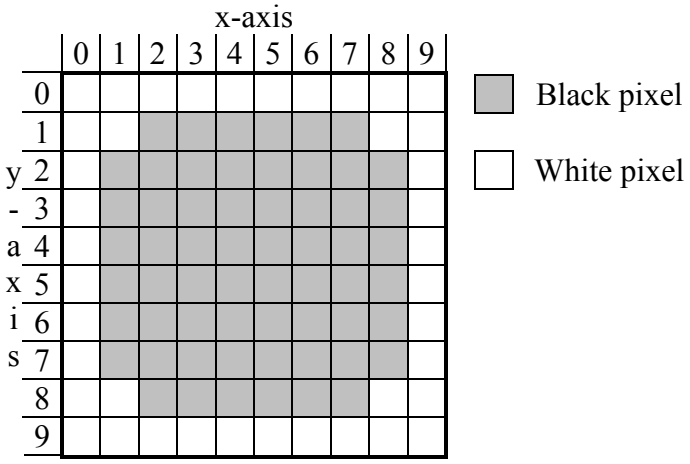


Figure 3.13 - A black and white 10 by 10 pixel image.

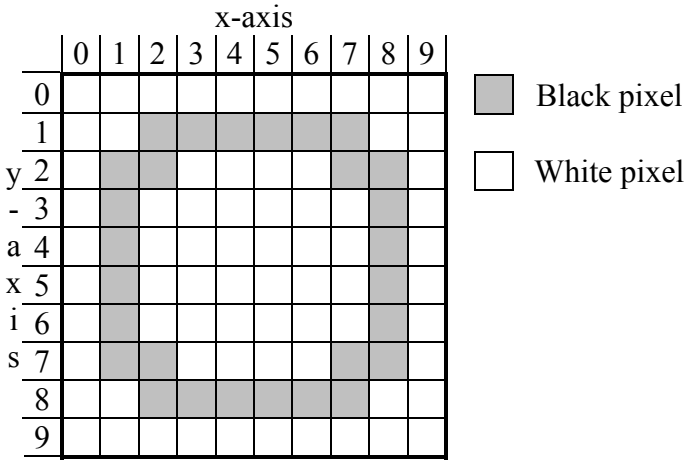


Figure 3.14 - A black and white 10 by 10 pixel image.

3.3.2 Computing the Convex Hull of a Connected Component

The computation of a connected component's convex hull uses the pixels from the connected component's outer border. The first step determines which outer border pixel is the pivot pixel. The pivot pixel is the pixel with the smallest y coordinate. If there is more than one pixel with the smallest y coordinate, the pivot pixel is the pixel with the smallest y coordinate and the smallest x coordinate.

Next, compute the polar coordinates, polar angle and distance, between each of the remaining outer border pixels and the pivot pixel. After computing the polar coordinates, sort the outer border pixels in ascending order according to their polar coordinates, first by their polar angle and then by their distance to the pivot pixel. For all pixels having the same polar angle, keep only the pixel with the furthest distance from the pivot pixel; remove all others from the list. This list of sorted pixels forms a queue.

The next part of the convex hull algorithm utilizes a pixel stack. The procedure begins by pushing the pivot pixel on top of the stack. Next, remove the pixel at the head

of the queue and push it on top of the stack. Finally, remove the pixel at the head of the queue and push it on top of the stack.

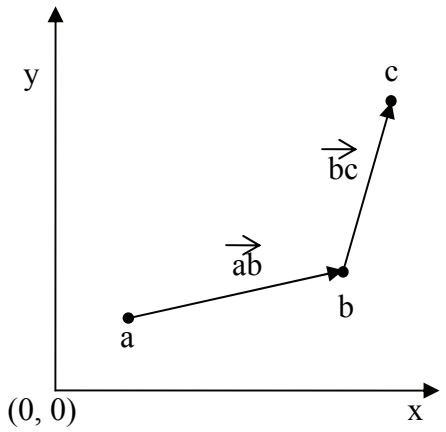


Figure 3.15 - The vectors forming a counter-clockwise movement.

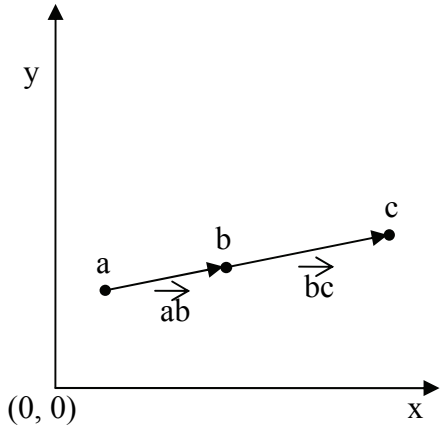


Figure 3.16 - The vectors forming a co-linear movement.

The algorithm executes the following two steps for each remaining pixel in the queue. Step 1 - remove the pixel at the head of the queue and label it *c*. Retrieve the pixel at the top of the stack labeling it *b* and retrieve the pixel immediately below the top of the stack labeling it *a*. Pixels *a* and *b* form the vector $\overrightarrow{ab}$ and the pixels *b* and *c* form the vector $\overrightarrow{bc}$. Step 2 - the movement from *a* to *b* to *c* along the vectors $\overrightarrow{ab}$ and $\overrightarrow{bc}$ take

one of three forms: counter-clockwise, co-linear and clockwise. Figure 3.15 demonstrates the two vectors having a counter-clockwise movement. Figure 3.16 demonstrates the two vectors having a co-linear movement. Figure 3.17 demonstrates the two vectors having a clockwise movement. The movement along the two vectors must follow the same form; in the case of this algorithm it is counter-clockwise. If the vectors $\overrightarrow{ab}$ and $\overrightarrow{bc}$ follow a co-linear or clockwise movement, pop a pixel off the top of the stack, retrieve the pixel at the top of the stack labeling it *b* and retrieve the pixel immediately below the top of the stack labeling it *a* to form new versions of the vectors $\overrightarrow{ab}$ and $\overrightarrow{bc}$. Popping pixels off the top of the stack forming new versions of vectors $\overrightarrow{ab}$ and $\overrightarrow{bc}$ continue until the vectors form a counter-clockwise movement. At that time, push the pixel *c* on top of the stack. Continue steps 1 and 2 until the queue is empty.

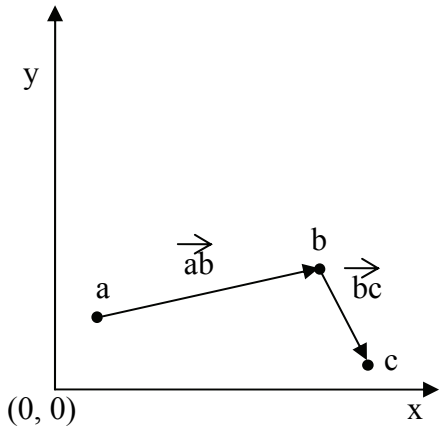


Figure 3.17 - The vectors forming a clockwise movement.

Once processing of all of the pixels in the queue completes, the pixels in the stack form the connected component's convex hull. Pop the pixels off the stack and place them in the convex hull linked list kept in the connected component's instance in the array *components*. Figure 3.18 illustrates a convex hull surrounding a connected component.

Figure 3.18 - The convex hull surrounding the character f.

3.4 The Genetic Algorithm to Recognize the Lines in the Image

The second phase in the OCR and CDC programs uses a genetic algorithm to determine the number of lines of text in the image as well as the position of each line in the image. The line recognition genetic algorithm evolves a population of lines, searching for a line whose slope closely matches the slope of the lines of text in the image. Figure 3.19 displays the flowchart of the line recognition genetic algorithm with section 5.2 in chapter 5 providing a detailed description of the line recognition genetic algorithm method. When the line recognition genetic algorithm terminates, the fitness value of the fittest individual provides the number of lines in the image.

The OCR and CDC programs use the random number generator coded by Ladd (1995) with subsection 5.2.2 in chapter 5 providing a detailed description of the two methods to generate integer and floating point random numbers for the OCR and CDC programs. The discussion begins, in the next subsection 3.4.1, with a description of the structures needed by the line recognition genetic algorithm. Subsection 3.4.2 describes the genetic algorithm's fitness function. Finally, subsection 3.4.3 details all of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation.

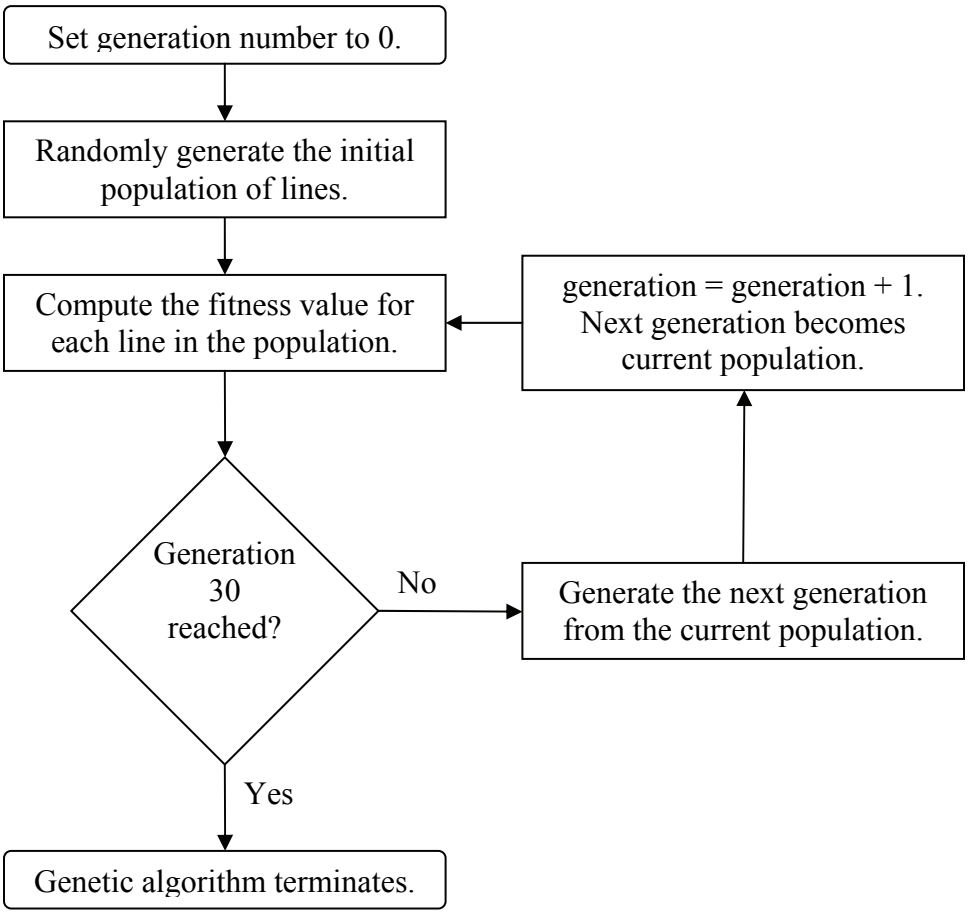


Figure 3.19 - Flowchart of the line recognition genetic algorithm.

3.4.1 Genetic Algorithm Structures Required for Line Recognition

The basic structure of the line recognition genetic algorithm is the centroid of each connected component in the image. Some of the connected components in the image due to their small size or unusual dimensions (corresponding to stray marks, dirt and characters such periods, commas, and apostrophes) may have an adverse affect on the convergence of the genetic algorithm to a reasonable solution, too few or too many lines that exist in the image. Therefore, a pre-processing step executes at the beginning of the genetic algorithm.

The pre-processing step of the genetic algorithm essentially divides up the connected components into two sets, one called gaComponents and the other called nonGAComponents. Place each connected component in gaComponents or nonGAComponents based on the following three threshold values. The area threshold is one fourth the average area for the connected component's minimum area rectangle. The width threshold is one third the average width for the connected component's minimum area rectangle. The height threshold is one third average height for the connected component's minimum area rectangle. For each connected component, if its minimum area rectangle area is greater than or equal to the area threshold and its minimum area rectangle width is less than the width threshold and its minimum area rectangle height is less than the height threshold, place the connected component in the gaComponent set; otherwise place the connected component in the nonGAComponent set.

The second structure of the line recognition genetic algorithm is the individual of the genetic algorithm population. The centroid points of two randomly chosen connected components of the gaComponent set forms each individual in this genetic algorithm's population. The two points of the individual form a line but to the genetic algorithm the line actually represents all lines with the same slope. The fitness function uses this representation of an individual to compute the individual's fitness value. The next subsection discusses the function to compute the individual's fitness value.

3.4.2 The Fitness Function

An individual's fitness value represents the number of lines in the image in which each of these lines has the same slope as the individual's line. The fitness function

requires two threshold values to control its execution. The first threshold value is the distance threshold used to determine if a connected component's centroid point is within a certain distance from a line. Compute the average height and the average width of the minimum area rectangle for the connected components in the gaComponents set. Multiply the larger of the two values by the distance threshold percentage of 71% to generate the distance threshold value. A varied set of test images helped tweak the threshold percentage from an initial value of 60% to its current value of 71%.

The second threshold value is the fitness threshold used to spot individuals that have very low fitness. Estimate the average number of rows in the image by dividing the height of the image by the average height of the minimum area rectangle for the connected components in the gaComponents set. Estimate the average number of columns in the image by dividing the width of the image by the average width of the minimum area rectangle for the connected components in the gaComponents set. The fitness threshold value is the sum of the average number of rows and the average number of columns in the image.

The text orientation parameter specifies the orientation vector used by the fitness function. If the value of the text orientation parameter is portrait then the orientation vector is the unit vector along the x-axis. If the value of the text orientation parameter is landscape then the orientation vector is the unit vector along the y-axis.

The fitness function begins by initializing the fitness value of the individual to zero. Create a Boolean array called processedComponents whose size is equal to the number of connected components in the gaComponents set where each element in processedComponents corresponds to a unique connected component in the

gaComponents set. Initialize each element in processedComponents to false meaning the fitness function did not process each connected component yet. Next, compute the vector V formed by the individual's two points.

Compute the skew angle between the individual's vector V and the orientation vector, see figure 3.20. It is reasonable to assume that the lines of text in the image will not be askew greater than 30 degrees. Therefore, an individual whose skew angle is greater than 30 degrees is an unfit individual automatically given an unfit fitness value. Hence, if the skew angle is greater than 30 degrees, the fitness value becomes the fitness threshold multiplied by 1,000.

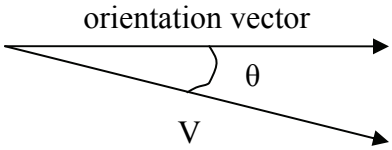


Figure 3.20 - Skew angle θ between the orientation vector and vector V.

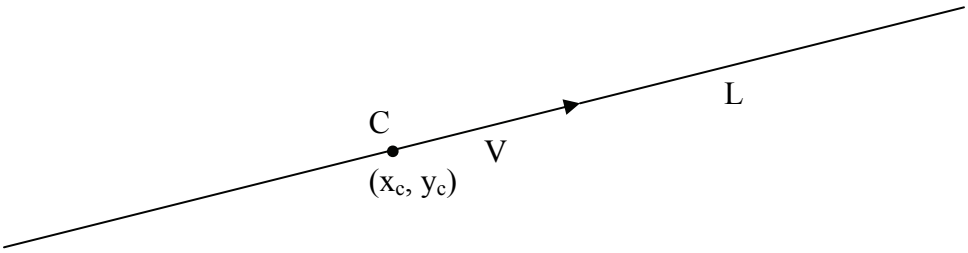


Figure 3.21 - Line L formed by the individual's vector V and a connected component's centroid point C.

Otherwise, if the skew angle is less than or equal to 30 degrees, the fitness function executes the following steps until it processes all of the connected components in gaComponents. Step 1: scan the processedComponents array to find the first unprocessed connected component C. Step 2: set C's corresponding element in

processedComponents to true to mark C as processed. Step 3: retrieve C's centroid point. Step 4: use C's centroid point as the endpoint of the vector V to form an actual line L in the image, see figure 3.21.

Step 5: scan processedComponents, for each unprocessed connected component C1, compute the distance between C1's centroid point and its projection onto the line L. If the distance is less than or equal to the distance threshold then set C1's corresponding element in processedComponents to true to mark C1 as processed. Figure 3.22 illustrates a number of unprocessed connected component centroid points near the current line L. Figure 3.23 displays the distance from the connected component centroid point C1 to the line L.

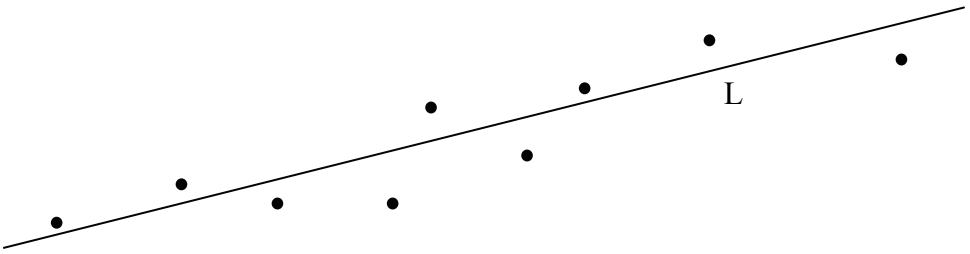


Figure 3.22 - Centroids from connected components near Line L.

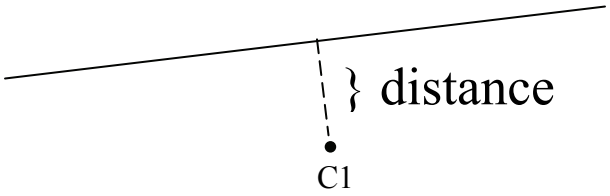


Figure 3.23 - Distance of Centroid C1 from the Line L.

Step 6 - Increase the individual's fitness value by 1. If there are any unprocessed connected components in processedComponents return to step 1 above. Once the fitness function processes all of the connected components in gaComponents, the fitness value

specifies the number of lines in the image for the individual. If the fitness value is greater than the fitness threshold, set the fitness value to the value of the fitness threshold multiplied by 1,000.

Finally, in this genetic algorithm, a smaller fitness value represents a fit individual and a larger fitness value represents a less fit individual. The roulette wheel selection method, described in the next subsection, makes the opposite assumption about the fitness value during its selection of an individual. Therefore, the fitness function returns the inverse of the computed fitness value. Once the fitness function computes the fitness value for each individual, the creation of the next generation begins according to the description in the next subsection.

3.4.3 Reproduction of the Next Generation

The population size, called popSize, of the line recognition genetic algorithm is equal to the image file's area divided by the average area of the minimum area rectangle for the connected components in the gaComponents set. Each iteration of the genetic algorithm produces the same number of lines, popSize, for the new generation as the current generation. Two parent lines from the current population breed an offspring line for the population of the new generation. The roulette wheel selection method enables the genetic algorithm to randomly choose an offspring's parents. Subsection 3.4.3.1 defines and illustrates the use of the roulette wheel selection. After choosing the parents, the crossover operator creates the offspring from portions of each parent. Subsection 3.4.3.2 details the application of the crossover operator. Finally, subsection 3.4.3.3

describes the application of mutation to each offspring which introduces variety into the genome of the lines.

3.4.3.1 The Roulette Wheel Selection Method

The genetic algorithm roulette wheel is a genetic algorithm selection method based on a gambler's roulette wheel to randomly choose an individual from the population. Thirty-eight equal sized sections divide the gambler's roulette wheel with 37 sections numbered 0 through 36 and the 38^{th} section numbered 00. The croupier spins the wheel in one direction and throws the marble in the opposite direction. Eventually the wheel and the marble stop moving with the marble coming to rest in one of the sections. This concept translates to a selection method in a genetic algorithm.

Each line in the population occupies one section of the genetic algorithm roulette wheel where the line's fitness value determines the size of the section. The genetic algorithm equivalent of the roulette wheel marble, generated for each selection of an individual from the population, is a randomly chosen integer between 0 and the sum total of the population's fitness values. Starting with the population's first individual, simulation of the motion of the wheel and marble executes the following two steps. Step 1 - is the marble's value less than the fitness value of the current individual? Step 2 - if the answer to the question in step 1 is yes then choose the current individual otherwise subtract the current individual's fitness value from the marble's value and repeat both steps for the next member of the population. Members of the population with higher fitness values have a higher chance of selection.

For example, consider the small example population in figure 3.24 listing each line's fitness value. Assume 54 is the roulette wheel marble; a randomly chosen value from the range of 0 to 67, the sum total of the population's fitness values from figure 3.24. Begin with line 1 in figure 3.24. According to step 1, the marble's value 54 is not less than line 1's fitness value 19. In this case, step 2 instructs the roulette wheel to subtract line 1's fitness value 19 from the marble's value 54 to make the marble's value 35 and to repeat steps 1 and 2 for the next member of the population, line 2. According to step 1, the marble's value 35 is not less than line 2's fitness value 25. In this case, step 2 instructs the roulette wheel to subtract line 2's fitness value 25 from the marble's value 35 to make the marble's value 10 and to repeat steps 1 and 2 for the next member of the population, line 3. According to step 1, the marble's value 10 is not less than line 3's fitness value 7. In this case, step 2 instructs the roulette wheel to subtract line 3's fitness value 7 from the marble's value 10 to make the marble's value 3 and to repeat steps 1 and 2 for the next member of the population, line 4. According to step 1, the marble's value 3 is less than line 4's fitness value 14. In this case, step 2 instructs the roulette wheel to choose line 4 as the individual selected. The next subsection describes the application of crossover to create an offspring line from two parent lines.

Individual	**Fitness Value**
line 1	19
line 2	25
line 3	7
line 4	14
line 5	2
Sum total of the population's fitness values: 67	

Figure 3.24 - An example population.

3.4.3.2 The Crossover Operator

Crossover is a genetic algorithm operator that creates an offspring from two parents. The roulette wheel selection method randomly chooses two individuals from the population to become the "parents" to produce an offspring. The line with the smallest fitness value is parent 1 and the other line is parent 2. After selection of both parents, crossover creates an offspring by mixing values from each parent according to a crossover rate. The crossover rate supplied to the line recognition genetic algorithm is 7%, a percentage between 0% and 100%, converted to the real number 0.07, a value between 0 and 1. Crossover applies to each endpoint of the offspring.

Randomly generate a real number x between 0 and 1. If x is greater than the crossover rate then the first endpoint of the offspring becomes the first endpoint of parent 1. Otherwise, the first endpoint of the offspring becomes the first endpoint of parent 2.

Randomly generate a second real number y between 0 and 1. If y is greater than the crossover rate then the second endpoint of the offspring becomes the second endpoint of parent 1. If both endpoints of the offspring are the same then the second endpoint of the offspring changes to the second endpoint of parent 2. If y is less than or equal to the crossover rate then the second endpoint of the offspring becomes the second endpoint of parent 2. If both endpoints of the offspring are the same then the second endpoint of the offspring changes to the second endpoint of parent 1. The next subsection describes the application of mutation to an offspring line after crossover creates the line.

3.4.3.3 The Mutation Operator

The mutation rate supplied to the line recognition genetic algorithm is 3%, a percentage between 0% and 100%, converted to the real number 0.03, a value between 0 and 1. Mutation applies to each endpoint of the offspring. Randomly generate a real number x between 0 and 1. If x is greater than the mutation rate then mutate the first endpoint of the offspring. Randomly choose a connected component from the gaComponents set such that the centroid point of the connected component is not equal to either endpoint of the offspring. Set the first endpoint of the offspring to the centroid point of the chosen connected component.

Randomly generate a second real number y between 0 and 1. If y is greater than the mutation rate then mutate the second endpoint of the offspring. Randomly choose a connected component from the gaComponents set such that the centroid point of the connected component is not equal to either endpoint of the offspring. Set the second endpoint of the offspring to the centroid point of the chosen connected component. In the next section, the fittest individual facilitates the creation of structures for each line in the image.

3.5 Organization of Components by Line

Organization of all of the connected components in the image according to their line of text begins after the fitness function determines the number of lines of text in the image. Subsection 3.5.1 describes the method to create the set of lines and assign each connected component to the appropriate line. Most characters consist of a single

connected component but some are made up of 2 or more connected components, certain ones by design such as ! or %, or others because of resolution issues. Subsection 3.5.2 describes the method to merge neighboring connected components together.

3.5.1 Creation and Organization of Each Line

The fitness function provides the fittest individual and its fitness value whose inverse is the total number of lines in the image. The findLines function utilizes the following values and structure to create and organize the lines in the image. The fittest individual's two points form the vector V. The two sets gaComponents and nonGAComponents list all of the connected components in the image. The findLines function creates the Boolean arrays processedGAComps and processedNonGAComps whose size is equal to the number of connected components in the gaComponents and nonGAComponents sets, respectively. Each element in processedGAComps and processedNonGAComps corresponds to a unique connected component in the gaComponents and nonGAComponents sets, respectively. Initialize each element in processedGAComps and processedNonGAComps to false meaning that the findLines function did not process the connected components yet. The term "axis intercept" designates the particular axis the lines in the image intercept, given by the text orientation parameter. If the value of the text orientation parameter is portrait then the lines intercept the y-axis; otherwise the value of the text orientation parameter is landscape meaning the lines intercept the x-axis.

The Line class stores a line and its values. A one-dimensional array of Line, called theLines, holds all of the lines in the image. Each element of the array theLines is

an instance of the Line class corresponding to one of the lines in the image. Each instance of the Line class stores four values in the following Line class data members. The data member numComps stores the number of components on that line. The data member lineOffset stores the offset or distance between the line's axis intercept and the origin (0, 0). The data member endPoints, which is a two element array of the Point class, stores the two points that specify the line itself. The Point class stores the x and y coordinates of a point. A non-existent line initializes each element in the array theLines.

The LineComp class stores information about a connected component and the connected component's relative position on its line. Each instance of the LineComp class stores two values, for its particular connected component, in the following LineComp class data members. The Components class contains an array, called *components*, storing all of the connected components in the image. The data member compNum stores the index of the element in the array *components* corresponding to the connected component. The data member compOffset stores the offset or distance between the connected component's centroid point and the x or y intercept of the connected component's line.

Each line in the image has a linked list of LineComp class instances for the connected components belonging to that line. A one-dimensional array of linked lists, called theLineComps, holds the linked lists for all of the lines in the image. The i^{th} linked list in the array theLineComps stores all of the connected components for the i^{th} line in the array theLines. Each linked list in the array theLineComps is initially empty.

The findLines function uses the same distance threshold value as the fitness function. Recall from the fitness function description that the distance threshold determines if a connected component's centroid point is within a certain distance from a

line. Compute the average height and the average width of the minimum area rectangle for the connected components in the gaComponents set. Multiply the larger of the two by the distance threshold percentage of 71% to generate the distance threshold value.

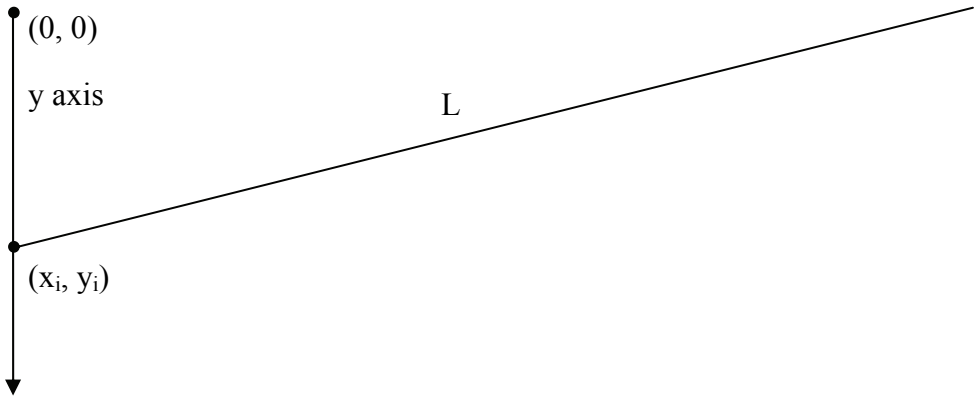


Figure 3.25 - Axis intercept point (x_i, y_i) of line L with the intercept axis y.

Creation of each line is a two step process. These two steps execute until the findLines function processes all of the connected components in the gaComponents set. Step 1 creates a new line and its values placing the line in the next available spot in the array theLines. Scan the processedGAComps array to find the first unprocessed connected component C. Retrieve the coordinates (x_c, y_c) of C's centroid point. Use C's centroid point as the endpoint of the vector V to form an actual line L in the image, see figure 3.21. Compute the coordinates (x_i, y_i) of the axis intercept point for the line L. Figure 3.25 illustrates the axis intercept point for a line L in an image whose intercept axis is the y axis. Find the next available element j in the array theLines to store the line L. The statement theLines[j].setLine(x_i, y_i, x_c, y_c) stores line L in element j of the array theLines. The four parameters to the method setLine are the coordinates of the two points that setLine stores in the data member endPoints. The method setLine initializes the data member numComps to zero because no connected components belong to the line

L, at this point. The method setLine computes the distance between the image's origin point (0, 0) and the axis intercept point (x_i, y_i) assigning it to the data member lineOffset.

In step 2, find all of the connected components that belong to the line L created in step 1. Scan processedGAComps, for each unprocessed connected component C, compute the distance between C's centroid point and its projection onto the line L. If the distance is less than or equal to the distance threshold then set C's corresponding element in processedGAComps to true to mark C as processed. Retrieve C's index from array *components* storing it in the variable compNum. Compute the distance between C's centroid point and line L's axis intercept point (x_i, y_i) storing it in the variable compOffset. Create an instance LC of the LineComp class for the connected component C. The statement LC.setLineComp(compNum, comOffset) calls the setLineComp method to store in the LineComp instance LC the two values of the connected component C. Recall from the previous paragraph, element j of the array theLines stores line L and the linked list element j of the array theLineComps stores all of the connected components belonging to line L. The statement theLineComps[j].insertSorted(LC) inserts the LineComp instance LC into the linked list in ascending sorted order according to LineComp's compOffset data member, LC's position on the line L relative to the line's axis intercept point. Process the connected components in the nonGAComponents set, here in step 2, in the same way as the connected components in the gaComponents set. The findLines function creates lines, executing steps 1 and 2, until it processes all of the connected components in the gaComponents set.

The arbitrary order of the lines in the theLines array requires the sortLines function to sort the lines in ascending order according to the line's lineOffset value, the

position of the line's intercept point relative to the image's origin (0, 0). The sortLines function uses the overloaded operator < of the Line class to sort the lines, theLines[i] < theLines[j]. The Line class operator < compares the lineOffset data member of theLines[i] to the lineOffset data member of theLines[j]. Since element i in the array theLineComps corresponds to element i in the array theLines, when the sort swaps two elements in the array theLines, the sort swaps the corresponding two elements in the array theLineComps. The findLines function computes the number of components on line i adding the value to the line's instance in the array theLines. The next subsection explains the final action of line organization.

3.5.2 Merging Certain Neighboring Components

There are 9 characters on the U.S. English keyboard that consist of two or more connected components. They are: the letters i and j, exclamation point, percent sign, equal sign, colon, semi-colon, double quotes and question mark. Also, if the scan resolution is low or the size of the characters is small, characters that normally should form a single connected component can have breaks in them that make the character consist of two or more components. The mergeComps function examines each connected component in the linked lists of the array theLineComps to merge neighboring components together when meeting a certain threshold distance. Compute the average width of the minimum area rectangle for all of the connected components in the image. Multiply the average width by the merge threshold percentage of 55% to generate the merge threshold value.

For each line's linked list in the array theLineComps, examine each connected component C in the list starting at the beginning of the list. The merge process is simple. Compute the projection of C's centroid point onto the line. If C has a left neighbor component in the linked list, compute the projection of the left neighbor's centroid point onto the line. Compute the distance between the two projection points. If the distance is less than or equal to the merge threshold, merge these two connected components together. Merging involves adding the pixels of the left neighbor to the connected component C and removing the left neighbor component from the linked list. If C has a right neighbor component in the linked list, compute the projection of the right neighbor's centroid point onto the line. Compute the distance between the two projection points. If the distance is less than or equal to the merge threshold, merge these two connected components together. Merging involves adding the pixels of the right neighbor to the connected component C and removing the right neighbor component from the linked list. Finally, re-compute the structures and statistical measures for connected component C, see section 3.3. The next section describes the CDC and OCR programs for the optical character recognition system.

3.6 The Two Optical Character Recognition System Programs

Both, the Character Database Creation (CDC) program and the Optical Character Recognition (OCR) program execute the same code until the end. The programs begin by opening the image file and finding all of the connected components in the image, see section 3.2. Then, both programs compute the statistics and structures for each connected component, see section 3.3. The line recognition genetic algorithm (see section 3.4)

determines the number of lines of text in the image. The creation and organization of the lines in the image (see section 3.5) is the next to the last piece of common code for both programs. The next subsection describes the last piece of common code that computes the values of the measures for each line's connected components. Subsection 3.6.2 describes the code specific to the character database creation program. Subsection 3.6.3 describes the code specific to the optical character recognition program.

3.6.1 Component Measure Computation

There are 19 measures for use in character recognition of each connected component in the image's lines. An array of 19 binary values, called measureSwitches, provides the CDC and OCR programs control over the use of each measure. Each element of the array corresponds to one of the measures where a value of 1 turns on the measure and a value of 0 turns off the measure. The first two measures are simple area measurements. The axis aligned area is the smallest rectangle enclosing the connected component in which the top and bottom sides of the rectangle are parallel to the x-axis of the image and the left and right sides of the rectangle are parallel to the y-axis of the image. Calculate the axis aligned area from the rectangle's width and height. The width is equal to x_{max} - x_{min} and the height is equal to y_{max} - y_{min}. Recall from section 3.3, x_{min}, y_{min}, x_{max} and y_{max} are values computed for the connected component. The second measure is the area of the connected component's minimum area rectangle.

$m_{pq} = \sum_x \sum_y x^p y^q f(x, y)$ where p, q = 0, 1, 2, …

$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y)$ where p, q = 0, 1, 2, …

$\bar{x} = m_{10} / m_{00}$
$\bar{y} = m_{01} / m_{00}$

$\eta_{pq} = \mu_{pq} / \mu_{00}^{\gamma}$ where $\gamma = 1 + (p + q) / 2$ for p+q = 2, 3, …

$\phi_1 = \eta_{20} + \eta_{02}$

$\phi_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2$

$\phi_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2$

$\phi_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2$

$$\phi_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

$$\phi_6 = (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}^2(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03})$$

$$\phi_7 = (3\eta_{21} - \eta_{30})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] + (3\eta_{12} - \eta_{30})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]$$

Figure 3.26 - The Hu moment equations.

The remaining 17 measures are statistical moments taken from the literature. Ming-Kuei Hu (1962) developed the first 7 moments presented in the paper by Celebi and Aslandogan (2005). Figure 3.26 lists all of the equations to compute the Hu moments. The symbol φ_i is the i^{th} Hu moment. For the binary image, f(x, y) = 1 when the pixel at coordinates (x, y) is black and f(x, y) = 0 when the pixel at coordinates (x, y) is white. Flusser and Suk (2004) developed the remaining 10 moments. Figure 3.27 lists all of the equations to compute the Flusser-Suk moments. The symbol I_i is the i^{th} Flusser-

Suk moment. For the binary image, f(x, y) = 1 when the pixel at coordinates (x, y) is black and f(x, y) = 0 when the pixel at coordinates (x, y) is white. The next subsection describes the code specific to the character database creation program.

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y) \text{ where p, q = 0, 1, 2, …}$$

$$\bar{x} = m_{10} / m_{00}$$
$$\bar{y} = m_{01} / m_{00}$$

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y) \text{ where p, q = 0, 1, 2, …}$$

$$I_1 = (\mu_{20}\mu_{02} - \mu_{11}^2) / \mu_{00}^4$$

$$I_2 = (-\mu_{30}^2\mu_{03}^2 + 6\mu_{30}\mu_{21}\mu_{12}\mu_{03} - 4\mu_{30}\mu_{12}^3 - 4\mu_{21}^3\mu_{03} + 3\mu_{21}^2\mu_{12}^2) / \mu_{00}^{10}$$

$$I_3 = (\mu_{20}\mu_{21}\mu_{03} - \mu_{20}\mu_{12}^2 - \mu_{11}\mu_{30}\mu_{03} + \mu_{11}\mu_{21}\mu_{12} + \mu_{02}\mu_{30}\mu_{12} - \mu_{02}\mu_{21}^2) / \mu_{00}^7$$

$$\begin{aligned} I_4 = (&-\mu_{20}^3\mu_{03}^2 + 6\mu_{20}^2\mu_{11}\mu_{12}\mu_{03} - 3\mu_{20}^2\mu_{02}\mu_{12}^2 - 6\mu_{20}\mu_{11}^2\mu_{21}\mu_{03} - 6\mu_{20}\mu_{11}^2\mu_{12}^2 \\ &+ 12\mu_{20}\mu_{11}\mu_{02}\mu_{21}\mu_{12} - 3\mu_{20}\mu_{02}^2\mu_{21}^2 + 2\mu_{11}^3\mu_{30}\mu_{03} + 6\mu_{11}^3\mu_{21}\mu_{12} \\ &- 6\mu_{11}^2\mu_{02}\mu_{30}\mu_{12} - 6\mu_{11}^2\mu_{02}\mu_{21}^2 + 6\mu_{11}\mu_{02}^2\mu_{30}\mu_{21} - \mu_{02}^3\mu_{30}^2) / \mu_{00}^{11} \end{aligned}$$

$$I_5 = (\mu_{40}\mu_{04} - 4\mu_{31}\mu_{13} + 3\mu_{22}^2) / \mu_{00}^6$$

$$I_6 = (\mu_{40}\mu_{22}\mu_{04} - \mu_{40}\mu_{13}^2 - \mu_{31}^2\mu_{04} + 2\mu_{31}\mu_{22}\mu_{13} - \mu_{22}^3) / \mu_{00}^9$$

$$I_7 = (\mu_{20}^2\mu_{04} - 4\mu_{20}\mu_{11}\mu_{13} + 2\mu_{20}\mu_{02}\mu_{22} + 4\mu_{11}^2\mu_{22} - 4\mu_{11}\mu_{02}\mu_{31} + \mu_{02}^2\mu_{40}) / \mu_{00}^7$$

$$\begin{aligned} I_8 = (&\mu_{20}^2\mu_{22}\mu_{04} - \mu_{20}^2\mu_{13}^2 - 2\mu_{20}\mu_{11}\mu_{31}\mu_{04} + 2\mu_{20}\mu_{11}\mu_{22}\mu_{13} + \mu_{20}\mu_{02}\mu_{40}\mu_{04} \\ &- 2\mu_{20}\mu_{02}\mu_{31}\mu_{13} + \mu_{20}\mu_{02}\mu_{22}^2 + 4\mu_{11}^2\mu_{31}\mu_{13} - 4\mu_{11}^2\mu_{22}^2 - 2\mu_{11}\mu_{02}\mu_{40}\mu_{13} \\ &+ 2\mu_{11}\mu_{02}\mu_{31}\mu_{22} + \mu_{02}^2\mu_{40}\mu_{22} - \mu_{02}^2\mu_{31}^2) / \mu_{00}^{10} \end{aligned}$$

$$\begin{aligned} I_9 = (&\mu_{30}^2\mu_{12}^2\mu_{04} - 2\mu_{30}^2\mu_{12}\mu_{03}\mu_{13} + \mu_{30}^2\mu_{03}^2\mu_{22} - 2\mu_{30}\mu_{21}^2\mu_{12}\mu_{04} + 2\mu_{30}\mu_{21}^2\mu_{03}\mu_{13} \\ &+ 2\mu_{30}\mu_{21}\mu_{12}^2\mu_{13} - 2\mu_{30}\mu_{21}\mu_{03}^2\mu_{31} - 2\mu_{30}\mu_{12}^3\mu_{22} + 2\mu_{30}\mu_{12}^2\mu_{03}\mu_{31} + \mu_{21}^4\mu_{04} \\ &- 2\mu_{21}^3\mu_{12}\mu_{13} - 2\mu_{21}^3\mu_{03}\mu_{22} + 2\mu_{21}^2\mu_{12}\mu_{03}\mu_{31} + \mu_{21}^2\mu_{03}^2\mu_{40} - 2\mu_{21}\mu_{12}^3\mu_{31} \\ &- 2\mu_{21}\mu_{12}^2\mu_{03}\mu_{40} + \mu_{12}^4\mu_{40}) / \mu_{00}^{13} \end{aligned}$$

$$I_{10} = (-\mu_{50}^2\mu_{05}^2 + 10\mu_{50}\mu_{41}\mu_{14}\mu_{05} - 4\mu_{50}\mu_{32}\mu_{23}\mu_{05} - 16\mu_{50}\mu_{32}\mu_{14}^2 + 12\mu_{50}\mu_{23}^2\mu_{14} - 16\mu_{41}^2\mu_{23}\mu_{05} - 9\mu_{41}^2\mu_{14}^2 + 12\mu_{41}\mu_{32}^2\mu_{05} + 76\mu_{41}\mu_{32}\mu_{23}\mu_{14} - 48\mu_{41}\mu_{23}^3 - 48\mu_{32}^3\mu_{14} + 32\mu_{32}^2\mu_{23}^2) / \mu_{00}^{14}$$

Figure 3.27 - The Flusser-Suk moment equations.

3.6.2 The Character Database Creation Program

At this point, the execution of the Character Database Creation (CDC) program differs from the Optical Character Recognition (OCR) program. The CDC program receives as parameters the filename of the image file (inFilename), the font name (fontName) and font size (fontSize) of the text in the image file. The image file contains the set of all printable characters in a particular font name and font size combination. The keyFilename parameter stores the name of the text file containing the character key for the image file. Each line in the key file is the corresponding line in the image. The characters in the key file line are the same characters and in the same order as in the corresponding line in the image. The CDC program reads the characters in the key file into a two-dimensional array of char, called theKey, where the number of rows correspond to the number of lines and the number of columns correspond to the number of characters on the line having the most characters. The preprocess parameter is a binary value to specify whether the CDC program performs preprocessing, value of 1, or adds the characters to the database, value of 0.

Preprocessing ensures that every connected component in the image corresponds to one of the 94 characters found on the U.S. English keyboard. Smudges and stray marks can appear in an image file when scanning a page of text. For each connected

component in the image, print the component's width, height and centroid point to a text file. The fontName and fontSize parameters form the name of the text file. For example, if the font name is Arial and the font size is 12 then the filename is Arial12.txt. The user searches the image file for each connected component listed in the text file. If the connected component does not correspond to one of the U.S. English characters then the user erases the connected component from the image file. A second preprocessing run ensures that every connected component in the image corresponds to one of the 94 characters found on the U.S. English keyboard.

After the preprocessing completes, the user sets the preprocessing parameter to 0 in order to add the connected components (characters) to the character database. The character database is a Microsoft Access file called OCR.mdb. Simple C++ wrapper classes for win32 ODBC function calls written by Vijay Mathew Pandyalakal, downloaded from the Code Project website www.codeproject.com, allow access to OCR.mdb within a C++ program. These classes essentially execute SQL statements, formed within the C++ program, on the database with any results (database records) sent back to the C++ program.

The CDC program inserts a new record in the Characters table in the OCR.mdb database file for each connected component in each linked list in the array theLineComps. Figure 3.28 displays the structure of the Characters table listing each field's name and data type. The index of the linked list element in the array theLineComps is the line number for all of the connected components in the linked list. Remember from subsection 3.5.1, the line's linked list maintains the connected components in sorted order reading from left to right. A connected component's relative

position within its linked list is its position number on the line. The CDC program retrieves the character from the array theKey file based on the connected component's line number and position number. The key file character is the value for the Character field of the connected component's database record.

Field Name	Data Type
Character	Text
Number	Number
Filename	Text
FontName	Text
FontSize	Number
AxisAlignedArea	Number
MinimumArea	Number
HuMoment01	Number
HuMoment02	Number
HuMoment03	Number
HuMoment04	Number
HuMoment05	Number
HuMoment06	Number
HuMoment07	Number
FlusserSukMoment01	Number
FlusserSukMoment02	Number
FlusserSukMoment03	Number
FlusserSukMoment04	Number
FlusserSukMoment05	Number
FlusserSukMoment06	Number
FlusserSukMoment07	Number
FlusserSukMoment08	Number
FlusserSukMoment09	Number
FlusserSukMoment10	Number

Figure 3.28 - The structure of the Character table in the OCR.mdb database file.

The Components class contains an array, called *components*, storing all of the connected components in the image. The index of the connected component's element in the array *components* (component's number) is the value for the Number field of the connected component's database record. The fontName and fontSize parameters are the

values for the FontName and FontSize fields, respectively, of the connected component's database record. The CDC program draws the connected component to a TIFF image having the same width and height as the connected component saving the TIFF image to a file whose filename has the following format. The fontName followed by the fontSize ending with the component's number. For example, if the font name is Arial, the font size is 12 and the component's number is 20 then the filename is Arial12_020.tif. The filename is the value for the Filename field of the connected component's database record. The remaining fields in the Characters table correspond to each of the 19 measures. The value of each of the connected component's measures is the value for the corresponding field of the connected component's database record. An SQL Insert statement adds the connected component's database record into the Characters table.

The CDC program can process several image files containing the set of all printable characters in other font name and font size combinations. After processing all of the image files, the CDC program calculates/re-calculates the properties for each font name and font size combination in the Characters table. The OCR.mdb database file stores these properties in the Properties table. Figure 3.29 displays the structure of the Properties table listing each field's name and data type. Each record in the Properties table contains the font name, font size, and the smallest, largest and average value for each measure field listed in the Characters table. An SQL Insert statement adds the font name and font size combination's record into the Properties table.

Field Name	Data Type
FontName	Text
FontSize	Number
SmallestAxisAlignedArea	Number
LargestAxisAlignedArea	Number
AverageAxisAlignedArea	Number
SmallestMinimumArea	Number
LargestMinimumArea	Number
AverageMinimumArea	Number
SmallestHuMoment01	Number
LargestHuMoment01	Number
AverageHuMoment01	Number
SmallestHuMoment02	Number
LargestHuMoment02	Number
AverageHuMoment02	Number
SmallestHuMoment03	Number
LargestHuMoment03	Number
AverageHuMoment03	Number
SmallestHuMoment04	Number
LargestHuMoment04	Number
AverageHuMoment04	Number
SmallestHuMoment05	Number
LargestHuMoment05	Number
AverageHuMoment05	Number
SmallestHuMoment06	Number
LargestHuMoment06	Number
AverageHuMoment06	Number
SmallestHuMoment07	Number
LargestHuMoment07	Number
AverageHuMoment07	Number

Field Name	Data Type
SmallestFlusserSukMoment01	Number
LargestFlusserSukMoment01	Number
AverageFlusserSukMoment01	Number
SmallestFlusserSukMoment02	Number
LargestFlusserSukMoment02	Number
AverageFlusserSukMoment02	Number
SmallestFlusserSukMoment03	Number
LargestFlusserSukMoment03	Number
AverageFlusserSukMoment03	Number
SmallestFlusserSukMoment04	Number
LargestFlusserSukMoment04	Number
AverageFlusserSukMoment04	Number
SmallestFlusserSukMoment05	Number
LargestFlusserSukMoment05	Number
AverageFlusserSukMoment05	Number
SmallestFlusserSukMoment06	Number
LargestFlusserSukMoment06	Number
AverageFlusserSukMoment06	Number
SmallestFlusserSukMoment07	Number
LargestFlusserSukMoment07	Number
AverageFlusserSukMoment07	Number
SmallestFlusserSukMoment08	Number
LargestFlusserSukMoment08	Number
AverageFlusserSukMoment08	Number
SmallestFlusserSukMoment09	Number
LargestFlusserSukMoment09	Number
AverageFlusserSukMoment09	Number
SmallestFlusserSukMoment10	Number
LargestFlusserSukMoment10	Number
AverageFlusserSukMoment10	Number

Figure 3.29 - The structure of the Properties table in the OCR.mdb database file.

After computing the properties for each font name and font size combination, the CDC program calculates the tolerances for the properties in the Properties table. For each field in the Properties table beginning with the word “Average”, find the smallest and largest value for all of the records in the Properties table. For example, for the

HuMoment01 measure, find the smallest AverageHuMoment01 and the largest AverageHuMoment01 value for all of the records in the Properties table. The tolerance for HuMoment01 computes as ("the largest AverageHuMoment01" – "the smallest AverageHuMoment01"). The CDC program computes a tolerance for each of the 19 measures storing the tolerances in the Tolerance table in the OCR.mdb database file. Figure 3.30 displays the structure of the Tolerance table listing each field's name and data type. An SQL Insert statement adds the tolerance record into the Tolerance table. The next subsection describes the code specific to the optical character recognition program.

Field Name	Data Type
AxisAlignedArea	Number
MinimumArea	Number
HuMoment01	Number
HuMoment02	Number
HuMoment03	Number
HuMoment04	Number
HuMoment05	Number
HuMoment06	Number
HuMoment07	Number
FlusserSukMoment01	Number
FlusserSukMoment02	Number
FlusserSukMoment03	Number
FlusserSukMoment04	Number
FlusserSukMoment05	Number
FlusserSukMoment06	Number
FlusserSukMoment07	Number
FlusserSukMoment08	Number
FlusserSukMoment09	Number
FlusserSukMoment10	Number

Figure 3.30 - The structure of the Tolerance table in the OCR.mdb database file.

3.6.3 The Optical Character Recognition Program

The normal execution of the OCR program's character recognition phase on a connected component is a two step process. In the first step, the OCR program searches the database for the font name and font size combination that closely fits the component. In the second step, the character recognition restricts itself to only those characters in the database having the font name and font size combination found in step one. The value of the fontSwitch parameter allows the OCR program to control the execution of step one. During some experiments, the OCR program skips the first step in order to test the performance of the second (character recognition) step. Subsection 3.6.3.1 discusses the font name and font size recognition step executed when the fontSwitch parameter's value is 1. If fontSwitch's value is 0 then the OCR program skips the first step and uses the font name and font size combination, for the second (character recognition) step, supplied via the parameters fontName and fontSize, respectively. Subsection 3.6.3.2 discusses the character recognition step. The parameter outFilename stores the name of the text file to hold the recognized characters.

3.6.3.1 The Font Name and Font Size Recognition Step

The SQL Select statement "SELECT * FROM Tolerance" retrieves the record from the Tolerance table containing the tolerances for all 19 measures. The first step searches the database for the font name and font size combination that closely matches the font name and font size combination of the connected component. The search executes SQL Select statements on the Properties table using a range of values for each

measure. The search step computes the range for each measure using a percentage of that measure's tolerance value with an initial percentage of 5%.

For the following example, the value of HuMoment01 for a connected component is 417.935 and the tolerance of HuMoment01 is 377.719845. With a percentage of 5%, the tolerance percentage is 18.88599225 forming the range 399.04900775 to 436.82099225 for the HuMoment01 measure of the connected component. The value of HuMoment02 for a connected component is 15.1438 and the tolerance of HuMoment02 is 260.121549. With a percentage of 5%, the tolerance percentage is 13.00607745 forming the range 2.13772255 to 28.14987745 for the HuMoment02 measure of the connected component. Assume that the measureSwitches array parameter turns on, uses, HuMoment01 and HuMoment02, and turns off, doesn't use, the remaining 17 measures. Figure 3.31 illustrates the SQL Select statement to execute on the Properties table.

```
SELECT * FROM Properties WHERE HuMoment01 >= 399.04900775 AND
HuMoment01 <= 436.82099225 AND HuMoment02 >= 2.13772255 AND
HuMoment02 <= 28.14987745
```

Figure 3.31 - An example of an SQL Select statement to retrieve records from the Properties table in the OCR.mdb database file.

The Select statement may not return any records. If the Select statement returns zero records then increase the percentage, generate a new Select statement and execute the statement. A loop performs these steps repeatedly until a Select statement returns at least one record. Section 5.3 in chapter 5 provides a detailed description of the nested loops to search the database for the font name and font size combination that closely matches the font name and font size combination of the connected component.

For each record returned, compute the Euclidean distance between the measures of the record's font name and font size combination and the connected component's measures. Continuing the example, assume that the measureSwitches array parameter turns on, uses, HuMoment01 and HuMoment02, and turns off, doesn't use, the remaining 17 measures. Figure 3.32 is the equation to compute the Euclidean distance for this example. AverageHuMoment01 and AverageHuMoment02 are the measure values from the font name and font size combination's record and HuMoment01 and HuMoment02 are the measure values for the connected component. The font and font size combination for the connected component is the font name and font size combination record with the shortest Euclidean distance between it and the connected component. The next subsection describes the character recognition step.

$$\sqrt{(HuMoment01 - AverageHuMoment01)^2 + (HuMoment02 - AverageHuMoment02)^2}$$

Figure 3.32 - An example of computing the Euclidean distance between a connected component and a font name and font size combination from the Properties table in the OCR.mdb database.

3.6.3.2 The Character Recognition Step

The second step of character recognition searches the database for the character that closely matches the connected component with the search restricted to the font name and font size combination either found in the first step or supplied to the OCR program. The search executes SQL Select statements on the Character table using a range of values for each measure. The search step computes the range for each measure using a percentage of that measure's tolerance value with an initial percentage of 5%.

Continuing the example from subsection 3.6.3.1, assume that the measureSwitches array parameter turns on, uses, HuMoment01 and HuMoment02 and turns off, doesn't use, the remaining 17 measures. SmallestHuMoment01, LargestHuMoment01, SmallestHuMoment02 and LargestHuMoment02 are the measure values from the font name and font size combination's record to compute the tolerance for HuMoment01 as (LargestHuMoment01 - SmallestHuMoment01) and the tolerance for HuMoment02 as (LargestHuMoment02 - SmallestHuMoment02).

Assume that the font name "TimesNewRoman" and font size "30" combination is the closest matching combination for the connected component in the example. The value of HuMoment01 for the connected component is 417.935 and the tolerance of HuMoment01 is 1948.610226. With a percentage of 5%, the tolerance percentage is 97.4305113 forming the range 320.5044887 to 515.3655113 for the HuMoment01 measure of the connected component. The value of HuMoment02 for the connected component is 15.1438 and the tolerance of HuMoment01 is 4416.675514. With a percentage of 5%, the tolerance percentage is 220.8337757 forming the range -205.6899757 to 235.9775757 for the HuMoment02 measure of the connected component. Figure 3.33 illustrates the SQL Select statement to execute on the Characters table.

```
SELECT * FROM Characters WHERE FontName = 'TimesNewRoman' AND
FontSize = 30 AND HuMoment01 >= 320.5044887 AND HuMoment01 <= 515.3655113
AND HuMoment02 >= -205.6899757 AND HuMoment02 <= 235.9775757
```

Figure 3.33 - An example of an SQL Select statement to retrieve records from the Characters table in the OCR.mdb database file.

The Select statement may not return any records. If the Select statement returns zero records then increase the percentage, generate a new Select statement and execute the statement. A loop performs these steps repeatedly until a Select statement returns at least one record. Section 5.4 in chapter 5 provides a detailed description of the nested loops to search the database for the character that closely matches the connected component.

For each record returned, compute the Euclidean distance between the measures of the record's character and the connected component's measures. Continuing the example, assume that the measureSwitches array parameter turns on, uses, HuMoment01 and HuMoment02 and turns off, doesn't use, the remaining 17 measures. Figure 3.34 is the equation to compute the Euclidean distance for this example. HuMoment01 and HuMoment02 are the measure values from the character's record and CharHuMoment01 and CharHuMoment02 are the measure values for the connected component. The character recognized for the connected component is the character whose database record is the shortest Euclidean distance between it and the connected component. The next section describes the verification of this dissertation's implementation moment code.

$$\sqrt{(CharHuMoment01 - HuMoment01)^2 + (CharHuMoment02 - HuMoment02)^2}$$

Figure 3.34 - An example of computing the Euclidean distance between a connected component and a character from the Characters table in the OCR.mdb database file.

3.7 Verification of Moment Code

The verification step ensures that this dissertation's implementation of the Hu and Flusser-Suk moment equations in the OCR and CDC programs are correct. Since both

the OCR and CDC programs have the same implementation of the moment equations, the verification step uses just the OCR program. The OCR program processes a test image of text to print out the 17 moment values for each connected component in the test image. The author M. Emre Celebi implements the set of seven Hu moments listed in (Celebi and Aslandogan, 2005) using the C programming language. The incorporation of Celebi's moment code into the OCR program allows the OCR program to print out the seven Hu moment values for each connected component in the test image using Celebi's moment code. The author's Jan Flusser and Tomas Suk implement the set of ten Flusser-Suk moments listed in (Flusser and Suk, 2004) using MATLAB. Incorporation of Flusser-Suk's moment code into the OCR program requires a translation of the MATLAB moment code into its equivalent C programming code. After the translation, the insertion of the equivalent C code allows the OCR program to print out the ten Flusser-Suk moment values for each connected component in the test image using Flusser-Suk's implementation of the Flusser-Suk moment equations. A comparison of the values between the dissertation's implementation and the author's implementation of the moment equations provides a means to make any corrections needed to the dissertation's implementation of the moment equations. The next chapter describes the algorithmic implementations for the signature recognition problem.

Chapter 4 Signature Recognition Algorithmic Implementations

This chapter contains all of the C++ program code for the signature recognition program of the signature recognition problem discussed in chapter 2. The next section describes all of the parameters used by the signature recognition program. Section 4.2 provides a detailed description of the two methods to generate integer and floating point random numbers for the signature recognition program. Section 4.3 provides a detailed description of the actual implementation of an agent. Section 4.4 provides a detailed description of the methods to create paths for the data sets of the signature recognition program. Section 4.5 provides a detailed description of the methods to compute an agent's fitness value. Section 4.6 provides a detailed description of the methods to create the crossover points. Section 4.7 provides a detailed description of the signature recognition genetic algorithm method. Section 4.8 provides a detailed description of the signature recognition program's main method. Section 4.9 provides a detailed description of the batch file and method to execute the signature recognition program on a Windows computer. Section 4.10 provides a detailed description of the batch file and method to execute the signature recognition program on the computational cluster. Finally, section 4.11 provides a detailed description of the landscape program's main method.

4.1 The Signature Recognition Program Parameters

The file parameters.txt contains all but two of the program setup and control parameters, see figure 4.1. The gridRows and gridColumns parameters specify the number of rows and columns, respectively, in the grid. The value for gridRows and gridColumns is 200 to provide a sizeable enough grid to contain paths of a wide variety

of lengths. The numInputs, numOutputs and numStates parameters define the number of inputs, outputs and states, respectively, of the agents utilized in the program. The numInputs and numOutputs parameters each have a value of 3 according to the problem definition in section 2.0 in chapter 2. The numStates parameter is 32 giving an agent enough states to define a wide range of behaviors.

Parameter	Description	Value
gridRows	The number of rows in the grid.	200
gridColumns	The number of columns in the grid.	200
numInputs	Number of agent input.	3
numOutputs	Number of agent outputs.	3
numStates	Number of agent states.	32
popSize	Population size.	16,000
maxNumGen	Maximum number of generations.	125
numRuns	Number of times to run the genetic algorithm per path.	100
maxMoveMult	Number used in computing the maximum number of actions an agent can execute while computing its fitness.	5
crossoverRate	Crossover rate.	.07
mutationRate	Mutation rate.	.03
fileNum	The number used in the name of the file containing the paths and the file containing the output data.	A value between 1 and 100.
pathNum	The current path number	A value between 1 and 100.
initialRunNum	The initial run number for the current path.	A value between 1 and numRuns.

Figure 4.1 - The signature recognition program parameters.

The popSize parameter defines the size of the genetic algorithm population at 16,000 individuals to provide a diverse population of agents. It is conceivable that the genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm terminates if it doesn't find a solution by the time the generation number equals the value of the maxNumGen parameter which is 125. The

value of 125 for the maximum number of generations (maxNumGen) provides a reasonable chance for the genetic algorithm to converge.

At a minimum, the program executes the genetic algorithm once for each path provided as input. But, the numRuns parameter stipulates that the program executes the genetic algorithm, for each path, 100 times to ensure the results for each path are consistent. Computing the fitness of an agent determines how much of the path the agent consumes. Placed in the grid, the agent receives inputs and performs actions. Conceivably, the majority of the agents may not consume the entire path; a practical upper limit constrains the number of actions the agent executes when trying to consume the entire path. Multiplying the length of the path in the grid by the value of the maxMoveMult parameter (5) computes the limit. The crossover and mutation operators use the crossover and mutation rate parameters, respectively, to create an offspring agent from two parent agents. The crossover and mutation rate parameter values used in this program are one of the pairings of a crossover rate and a mutation rate: [5%, 6%, 7%, 8%, 9%, 10%]×[1%, 2%, 3%] chosen by the genetic algorithm tuning program discussed in subsection 4.7.2. In addition, the genetic algorithm tuning program also determines the optimal values for the program parameters numStates: value of 32, popSize: value of 16000, maxNumGen: value of 125, and maxMoveMult: value of 5.

The text file status.txt contains an arbitrary number of pairs of numbers with each pair on a separate line. In each pair, the first number is a path number and the second number is a run number such that the pair of numbers represents the run number of the genetic algorithm run completed for a particular path number. Hence, the last pair of numbers in status.txt is the last successfully completed run of the genetic algorithm for

the current path providing the initial value of the last two program parameters, pathNum and initialRunNum, respectively. The next section discusses the random number generator used by the signature recognition program.

4.2 Random Number Generator for the Signature Recognition Program

This signature recognition program uses the random number generator coded by Ladd (1995). The static class called Random incorporates Ladd's (1995) random number generator. Figure 4.2 lists the declaration of the Random class making up the header file Random.h. The Random class has one static member randNumGen (line 10, figure 4.2) of type RandDev. The class RandDev is the class written by Ladd (1995) defining his random number generator. The two methods of the Random class provide a convenient interface to Ladd's (1995) random number generator. The first method randomInt (line 6, figure 4.2) provides a means to randomly choose an integer from a range of integers. The second method randomFloat (line 7, figure 4.2) provides a means to randomly choose a floating point number from a range of floating point numbers.

```
 1: #include "RandDev.h"
 2:
 3: class Random
 4: {
 5:   public:
 6:     static int randomInt(int, int);
 7:     static float randomFloat(float, float);
 8:
 9:   private:
10:     static RandDev randNumGen;
11: };
```

Figure 4.2 - Declaration of the random number generator class.

Figure 4.3 lists the definition of the Random class making up the source code file Random.cpp. Line 1 in figure 4.3 includes the header file Random.h in the source file. The static member randNumGen (line 3, figure 4.3) creates an instance of data type RandDev initialized by RandDev's constructor.

```
 1: #include "Random.h"
 2:
 3: RandDev Random::randNumGen;
 4:
 5: int Random::randomInt(int sizeOfRange,
 6:                       int offsetOfRange)
 7: {
 8:   return ((int) (randNumGen() * sizeOfRange +
 9:                  offsetOfRange));
10: }
11:
12: float Random::randomFloat(float sizeOfRange,
13:                           float offsetOfRange)
14: {
15:   return (randNumGen() * sizeOfRange + offsetOfRange);
16: }
```

Figure 4.3 - Definition of the methods of the random number generator class.

The method randomInt (lines 5 to 10, figure 4.3) randomly chooses an integer from a particular range of integers. The first parameter of the method, sizeOfRange, defines the size of the range of integers. Ladd (1995) defined the () operator in his RandDev class to randomly generate a floating point value that is greater than or equal to 0 and less than 1. A call to this () operator takes the form of randNumGen() in the Random class. At line 8 in figure 4.3, the call to randNumGen() multiplied by the parameter sizeOfRange produces a randomly chosen value that is greater than or equal to 0 and less than sizeOfRange. For example, if sizeOfRange is 4, the randomly chosen value is greater than or equal to 0 and less than 4. The second parameter of the method,

offsetOfRange, specifies an offset value to allow the randomly chosen value to come from any possible range of integers. The randomInt method adds offsetOfRange to the result of multiplying randNumGen() by sizeOfRange as seen in lines 8 and 9 of figure 4.3. For example, if sizeOfRange is 4 and offsetOfRange is 6, the randomly chosen value is greater than or equal to 6 and less than 10. Since the randomly chosen value is a floating point value, a cast to int is necessary because the method randomInt returns a randomly chosen integer.

The method randomFloat (lines 12 to 16, figure 4.3) randomly chooses a floating point number from a particular range of floating point numbers. The first parameter of the method, sizeOfRange, defines the size of the range of floating point numbers. The second parameter of the method, offsetOfRange, specifies an offset value to allow the randomly chosen value to come from any possible range of floating point numbers. The only difference between the definition of randomFloat and randomInt is the cast to int not needed by randomFloat before returning the randomly chosen floating point number. The next section provides a description of the implementation of an agent.

4.3 Implementation of an Agent

In this signature recognition program, a one-dimensional array of integers actually implements the agent table with the number of elements in the array equal to numStates. Therefore, each element of the array represents one of the states of the agent. Figure 4.5 demonstrates how the output and nextState values for state 0 of the agent in figure 4.4a encode as an integer. The “binary” row in figure 4.5 displays the binary representation of

the integer encoding state 0, with the "positions" row in figure 4.5 displaying the bit positions of the binary representation.

States	Inputs S	Inputs E	Inputs C
0	L/23	A/5	L/25
1	L/22	R/19	A/1
2	L/5	A/14	A/18
3	R/18	R/19	A/30
4	A/18	A/28	A/21
5	R/18	L/0	R/26
6	L/4	R/24	L/1
7	A/24	L/4	R/23
8	A/26	L/4	A/28
9	L/18	A/26	L/18
10	L/1	R/0	A/15
11	A/26	R/2	R/0
12	A/24	R/14	R/23
13	R/13	R/22	L/7
14	A/19	L/23	R/26
15	R/8	R/14	R/0
16	L/5	A/21	A/1
17	A/18	A/24	R/17
18	A/18	R/17	R/30
19	R/13	L/11	L/28
20	A/3	R/17	L/14
21	R/20	A/28	R/23
22	R/22	A/17	A/0
23	A/2	R/28	R/12
24	R/12	R/23	A/7
25	L/12	L/15	L/10
26	R/4	A/21	R/14
27	L/13	L/2	R/0
28	A/16	R/7	L/12
29	A/15	L/6	L/31
30	A/7	R/22	R/11
31	L/2	A/0	R/10

(a) Un-encoded agent.

States	Integer encoded state
0	934583
1	27062
2	296741
3	502226
4	347666
5	1478738
6	551972
7	1430040
8	463386
9	822578
10	253985
11	1057050
12	1435416
13	650061
14	1481619
15	1058632
16	19109
17	1330194
18	1550482
19	988621
20	764035
21	1429076
22	2262
23	1256962
24	125900
25	694188
26	1280708
27	1052973
28	730000
29	1037071
30	1239815
31	1212450

(b) Encoded agent.

Figure 4.4 - An example of an agent encoded as integers.

	Unused bits	Input C's bits								Input E's bits								Input S's bits							
		Output		Next State						Output		Next State						Output		Next State					
Values:		L		25						A		5						L		23					
Positions:	31 through 21	20	19	18	17	16	15	14		13	12	11	10	9	8	7		6	5	4	3	2	1	0	
Binary:	00000000000	0	1	1	1	0	0	1		0	0	0	0	1	0	1		0	1	1	0	1	1	1	
Decimal:	934583																								

Figure 4.5 - An example of encoding state 0 in the agent of figure 4.4a as an integer.

Since the number of outputs is 3, the "output" value needs 2 bits. The outputs encode as: 00 for "A", 01 for "L", 10 for "R" with 11 unused. The methods that modify outputs in an agent are initializeFSM, crossover and mutate. The crossover method merely copies the existing valid output values from either parent to the offspring. The initializeFSM and mutate methods replace the existing valid output value with a new output value generated by the method chooseOutput. The chooseOutput method uses the random number generator to produce an integer that is greater than or equal to 0 and less than numOutputs. Since numOutputs is 3, the chooseOutput method will not chose the binary value 11 for an output in an agent.

Since the number of states is 32, the "next state" value needs 5 bits. The input "F" uses bits 0 through 6, the input "E" uses bits 7 through 13 and the input "C" uses bits 14 through 20, with the remaining bits 21-31 unused. Thus, an integer (32 bits) sufficiently encodes a given state. A total of 32 integers (one for each state) represent a given agent for the signature recognition program. The access methods getOutput, getNextState, setOutput, setNextState are the only methods to access the bits of a state of an agent and only manipulate bits 0 through 20 of the state. The row labeled "values" in figure 4.5 lists the output and next state values for state 0. The "decimal" row in figure 4.5 displays the decimal representation of the integer encoding state 0. Bit shifting and a

bit mask allow the methods to read or write an output or a next state value from an agent's state. Figure 4.4b displays the integer-encoded version of the un-encoded agent in figure 4.4a. The signature recognition program implements the agents in this way in order to minimize the memory requirements for the genetic algorithm population. The next section discusses the creation and storage of the program's paths.

4.4 Creation and Storage of the Paths

This section provides a detailed description of the methods to create paths for the data sets of the signature recognition program. The next subsection describes the main method createNewNonCrossoverPath to create the set of 10,000 non-crossover paths. Subsection 4.4.2 describes the main method createNewCrossoverPath to create the set of 10,000 crossover paths. The subsections 4.4.3 through 4.4.9 provide descriptions of the auxiliary methods used by each main method. Figure 4.6 defines the constants for use in any of the methods defined in this section.

The main path creation method creates a path one leg at a time and is a method of the Path class. A leg of the path represents one of the line segments of the path with the legs of the path stored in the order of creation. The direction of each new leg is perpendicular to the direction of the previous leg. Two values represent the leg, its length and the direction to move, in order to re-create the leg in the grid. The main method uses the seven Path class variables: thePath, pathStartX, pathStartY, pathLength, startFSMX, startFSMY and startFSMDirection to store all of the values defining a path. The variable thePath is a linked list structure used to store the legs of the path. The variables pathStartX and pathStartY contain the coordinates of the path's starting square. The

variable pathLength contains the length of the path. The variables startFSMX and startFSMY contain the coordinates of the agent's starting grid square. The variable startFSMDirection contains the direction the agent faces when placed in its starting grid square for the path.

```
#define NORTH 0
#define EAST  1
#define SOUTH 2
#define WEST  3

#define XDIR 0
#define YDIR 1

#define EMPTY  0
#define SQUARE 1

#define BORDER 2
#define NUMDIR 4
#define NUMDIM 2
```

Figure 4.6 - The constants used by the methods defined in section 4.4.

29 3 2 3 3 1 2 3 3 2 10 1 5 2 8 1 -1 -1

Figure 4.7 - The sequence of integers for the path in figure 4.8.

After path creation, the main path creation method writes, to the data file, the sequence of integers representing the path with each path kept on a separate line. Figure 4.7 illustrates the sequence of integers to store the path displayed in figure 4.8. The first number in the sequence, 29, is the path length of the path in figure 4.8. The second and third numbers (3, 2) are the (x, y) grid coordinates of the starting grid square of the path in figure 4.8. The fourth and fifth numbers (3, 3) in the sequence are the (x, y) grid coordinates of the starting grid square of the agent, always an EMPTY grid square next to the starting square of the path. The sixth number, 1, is the direction the agent faces when

placed in its starting grid square. The numbers 0, 1, 2 and 3 represent the directions north, east, south and west, respectively.

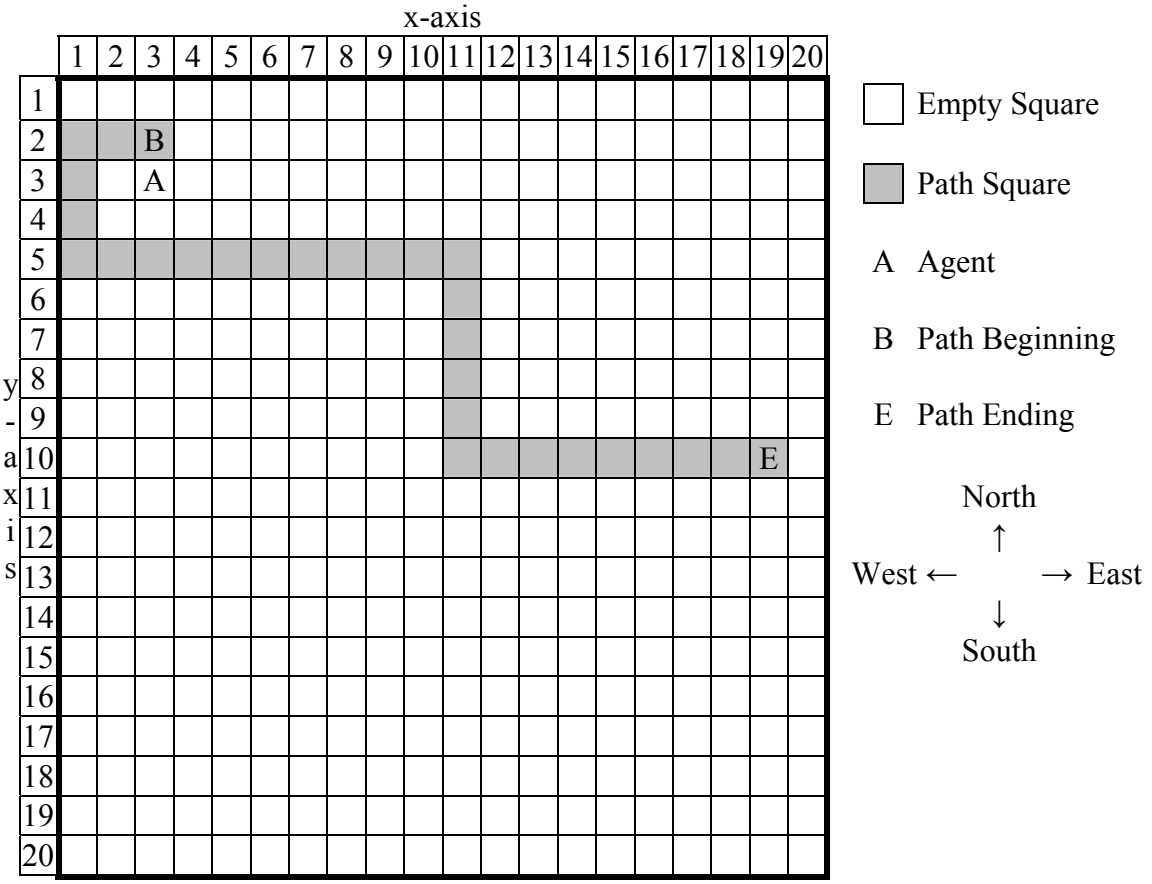


Figure 4.8 - A 20 x 20 grid containing a path and an agent.

Beginning with the seventh and eighth numbers in the sequence, each pair of numbers represents the length of the next leg of the path and the direction that the leg travels. The seventh and eighth numbers in the sequence, 2 and 3, are the length and direction of path leg 1. The ninth and tenth numbers in the sequence, 3 and 2, are the length and direction of path leg 2. The eleventh and twelfth numbers in the sequence, 10 and 1, are the length and direction of path leg 3. The thirteenth and fourteenth numbers in the sequence, 5 and 2, are the length and direction of path leg 4. The fifteenth and

sixteenth numbers in the sequence, 8 and 1, are the length and direction of path leg 5. The last two numbers (-1 -1) in the sequence, the seventeenth and eighteenth numbers, are flags to signal that there are no more legs for the path.

The signature recognition program reads in a path into an instance of the Path class. The data members of the program‘s instance of the Path class store the path length, the coordinates of the path’s starting square, the coordinates of the agent’s starting grid square, the agent’s starting direction and the legs of the path.

4.4.1 The Method createNewNonCrossoverPath

The two parameters at lines 1 and 2 in figure 4.9 of the method createNewNonCrossoverPath specify the number of rows in the grid, numRows, and the number of columns in the grid, numCols. The method createNewNonCrossoverPath begins at line 12 in figure 4.9 by clearing the linked list variable thePath of elements via a call to the makeEmpty method. The statement at line 13 in figure 4.9 allocates the storage for an array of numRows+BORDER elements where each element eventually holds a pointer to the storage allocated for one of the rows of the grid G. The call (line 14, figure 4.9) to the method allocateGridRows (see subsection 4.4.3) allocates the storage for each row of the grid, passing the number of grid rows, numRows, the number of grid columns, numCols, and the grid G as parameters. Next, the call (line 15, figure 4.9) to the method initializeGrid (see subsection 4.4.3) sets each square in the grid to EMPTY, passing the number of grid rows, numRows, the number of grid columns, numCols and the grid G as parameters. At line 16 in figure 4.9, a call to the randomInt method (see section 4.2) with parameter values numCols and 1 randomly chooses, from

the range 1 to numCols, the x coordinate of the path's starting square, pathStartX. At line 17 in figure 4.9, a call to the randomInt method (see section 4.2) with parameter values numRows and 1 randomly chooses, from the range 1 to numRows, the y coordinate of the path's starting square, pathStartY. At line 18 in figure 4.9, set the path's starting square, G[pathStartY][pathStartX], to the value SQUARE and initialize to 1 (line 19, figure 4.9) the pathLength variable.

```
 1: void Path::createNewNonCrossoverPath(int numRows,
 2:                                       int numCols)
 3: {
 4:   bool end, terminate;
 5:   int i, j, xPos, yPos, xAhead, yAhead;
 6:   int inc, direction, legLength;
 7:   int *G[];
 8:   int dirOffsets[NUMDIR][NUMDIM] = {{0, -1}, {1, 0},
 9:                                     {0, 1},  {-1, 0}};
10:   PathLeg pl;
11:
12:   thePath.makeEmpty();
13:   G = new int*[numRows+BORDER];
14:   allocateGridRows(numRows, numCols, G);
15:   initializeGrid(numRows, numCols, G);
16:   pathStartX = Random::randomInt(numCols, 1);
17:   pathStartY = Random::randomInt(numRows, 1);
18:   G[pathStartY][pathStartX] = SQUARE;
19:   pathLength = 1;
20:   xPos = pathStartX;
21:   yPos = pathStartY;
22:   direction = chooseDirection(numRows, numCols,
23:                               xPos, yPos);
24:   inc = chooseIncrement(numRows, numCols, direction,
25:                         xPos, yPos,);
26:   legLength = 0;
27:   end = false;
28:   do {
29:     terminate = false;
30:     for (i=1; i<=inc && !terminate; i++)
31:     {
32:       xAhead = xPos + dirOffsets[direction][XDIR];
33:       yAhead = yPos + dirOffsets[direction][YDIR];
34:       if (isSquareAheadClear(xAhead, yAhead,
```

```
35:                                direction, G))
36:       {
37:         xPos = xAhead;
38:         yPos = yAhead;
39:         G[yPos][xPos] = SQUARE;
40:         pathLength++;
41:         legLength++;
42:       }
43:       else
44:       {
45:         terminate = true;
46:       }
47:     }
48:     xAhead = xPos + dirOffsets[direction][XDIR];
49:     yAhead = yPos + dirOffsets[direction][YDIR];
50:     if (!(isSquareAheadClear(xAhead, yAhead,
51:                                direction, g)))
52:     {
53:       G[yPos][xPos] = EMPTY;
54:       pathLength--;
55:       legLength--;
56:       xPos = xPos - dirOffsets[direction][XDIR];
57:       yPos = yPos - dirOffsets[direction][YDIR];
58:     }
59:     pl.setPathLeg(legLength, direction);
60:     thePath.insertLast(pl);
61:     legLength = 0;
62:     end = doesPathEnd(xPos, yPos, direction, G);
63:     if (!end)
64:     {
65:       chooseNewDirAndInc(direction, inc, G,
66:                          xPos, yPos);
67:     }
68:   } while (!end);
69:
70:   chooseFSMStartPosition(numRows, numCols,
71:                          dirOffsets, G);
72:   startFSMDirection = Random::randomInt(NUMDIR, 0);
73:   deallocateGridRows(numRows, g)
74:   delete [] g;
75: }
```

Figure 4.9 - The main method to create a new non-crossover path.

The variables xPos and yPos hold the coordinates of the square just added to the path. The lines 20 and 21 in figure 4.9 assign xPos and yPos to the coordinates of the starting square of the path, pathStartX and pathStartY, respectively. The variable *direction* contains the direction of the current leg added to the path. The call (lines 22 and 23, figure 4.9) to the method chooseDirection (see subsection 4.4.4) assigns the direction of the first leg of the path to the variable *direction*, passing the number of grid rows, numRows, the number of grid columns, numCols, and the coordinates of the starting square of the path, xPos and yPos, as parameters. A call (lines 24 and 25, figure 4.9) to the method chooseIncrement (see subsection 4.4.4) assigns the variable *inc* the maximum possible length of the first leg of the path, passing the number of grid rows, numRows, the number of grid columns, numCols, the coordinates of the starting square of the path, xPos and yPos, and the direction of the first path leg as parameters. The legLength variable contains the current leg's length initialized to 0 at line 26 in figure 4.9. The variable *end* is a flag used to control the execution of the "do-while" loop of lines 28 through 68 in figure 4.9, initialized to false at line 27 in figure 4.9.

The "do-while" loop generates the legs of the path. The "for" loop (lines 30 to 47, figure 4.9) constructs a leg of the path one square at a time. The variable *i* is a counter variable, initialized to 1, controlling the execution of the "for" loop to terminate (line 30, figure 4.9) when *i*'s value is greater than the variable *inc*. But, the Boolean variable *terminate*, initialized to false at line 29 in figure 4.9, also controls the execution of the "for" loop possibly terminating the loop before *i*'s value is equal to *inc*. Therefore, the actual length of the current leg can be less than or equal to *inc*. Inside the body of the "for" loop at lines 32 and 33 in figure 4.9, the directional offsets in the current direction

added to the coordinates (xPos, yPos) of the previous square added to the leg computes the coordinates (xAhead, yAhead) of the next square to add to the leg. The "if" statement at lines 34 through 46 in figure 4.9 calls the method isSquareAheadClear (see subsection 4.4.5) to determine if the square at coordinates (xAhead, yAhead) in the grid G can add itself to the end of the current path leg. Lines 37 through 41 in figure 4.9 execute if the method isSquareAheadClear returns true. The lines 37 and 38 in figure 4.9 assign the coordinates xPos and yPos to the coordinates xAhead and yAhead, respectively. Line 39 in figure 4.9 sets the grid square at coordinates (xPos, yPos) to the value SQUARE. The pathLength and legLength variables increment by 1 (lines 40 and 41, figure 4.9) to reflect the square just added to the current path leg. If the method isSquareAheadClear returns false, another path square occupies the grid square at coordinates (xAhead, yAhead) ending the addition of any further squares to the current leg, line 45 in figure 4.9 sets the flag *terminate* to false.

After the "for" loop terminates, lines 48 and 49 in figure 4.9 compute the coordinates (xAhead, yAhead) of the next grid square past the end of the newly created leg. The "if" statement at lines 50 through 58 in figure 4.9 determines if the area around the coordinates (xAhead, yAhead) in the grid G are EMPTY, to ensure that the end of the newly added path leg is not immediately next to another part of the path. As an example, path leg 9 in figure 4.10 has direction EAST and ends at grid square G[9][11]. The method isSquareAheadClear (see subsection 4.4.5) examines the grid squares G[8][12], G[9][12] and G[10][12] and returns false because all three grid squares are not EMPTY. The not operator value receives the value returned by the call to the method isSquareAheadClear at lines 50 and 51 in figure 4.9. Since, "not" false is true, execute

lines 53 through 57 in figure 4.9. Line 53 in figure 4.9 sets the grid square at the end of the newly created path leg to EMPTY. Lines 54 and 55 in figure 4.9 decrements the variables pathLength and legLength by 1 to reflect the removal of the square at the end of the newly created path leg. Lines 56 and 57 in figure 4.9 subtract the directional offsets in the current direction from the coordinates (xPos, yPos) to compute the coordinates of the square that is now at the end of the newly created path leg. For the example in figure 4.10, the last square (xPos, yPos) of path leg 9 is now G[9][10].

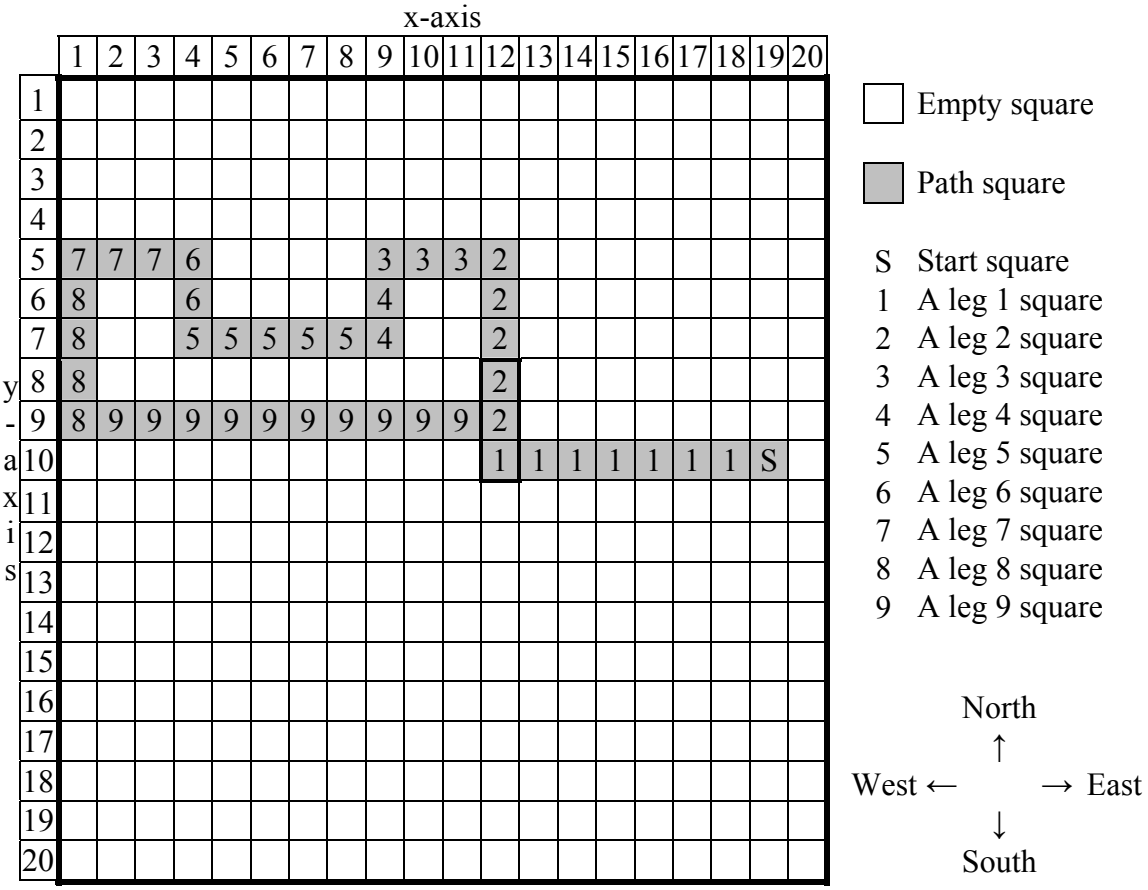


Figure 4.10 - A 20 x 20 grid containing a path.

The variable *pl* is an instance of the PathLeg class storing the path leg's length and direction. The call to the setPathLeg method (line 59, figure 4.9) of the PathLeg

class stores the values of the legLength and *direction* variables inside the PathLeg variable *pl*. A call to the insertLast method (line 60, figure 4.9) of the LinkedList class, passing the variable *pl* as a parameter, inserts a copy of the variable *pl* at the end of the LinkedList variable thePath. The legLength variable set to 0 at line 61 in figure 4.9 in order to get it ready to possibly add another leg to the path. A call to the method doesPathEnd (see subsection 4.4.6) assigns true to the variable *end* if the path can not add another path leg or false otherwise (line 62, figure 4.9), passing the direction, the coordinates (xPos, yPos) of the last square of the current path leg and the grid G as parameters. If the value of the variable *end* is false (line 63, figure 4.9) then a call (lines 65 and 66, figure 4.9) to the method chooseNewDirAndInc (see subsection 4.4.8) computes the direction and maximum possible length of the next path leg of the path, passing the direction, maximum possible length (*inc*) and coordinates (xPos, yPos) of the last square of the current path leg, the grid G and the value "true" as parameters. The direction and maximum possible length of the next path leg pass back out via the *direction* and *inc* parameters, respectively, due to pass by reference implementation. The body of the "do-while" loop (lines 28 to 68, figure 4.9) executes again if the value of the variable *end* is false (line 68, figure 4.9). Otherwise, the "do-while" loop terminates.

After the "do-while" loop terminates (lines 70 and 71, figure 4.9), a call to the method chooseFSMStartPostion (see subsection 4.4.9) selects the starting position of the agent for the newly created path, passing the number of rows and columns, numRows and numCols, respectively, the directional offsets, dirOffsets, and the grid G as parameters. At line 72 in figure 4.9, a call to the randomInt method (see section 4.2) with parameter values NUMDIR and 0 randomly chooses, from the range 0 to NUMDIR-1, the starting

direction of the agent, startFSMDirection. The call (line 73, figure 4.9) to the method deallocateGridRows (see subsection 4.4.3) de-allocates the storage for each row of the grid, passing the number of grid rows, numRows, and the grid G as parameters. The statement at line 74 in figure 4.9 de-allocates the storage for the array G of pointers to the storage allocated for each row of the grid. The next subsection describes the main method createNewCrossoverPath to create the set of 10,000 crossover paths.

4.4.2 The Method createNewCrossoverPath

The first two parameters (line 2, figure 4.11) of the createNewCrossoverPath method specify the number of rows in the grid, numRows, and the number of columns in the grid, numCols. The second two parameters (line 3, figure 4.11) of the createNewCrossoverPath method specify the smallest value, minPathLength, and the largest value, maxPathLength, for the range of values from which to randomly choose the path length. The method createNewCrossoverPath begins at line 12 in figure 4.11 by clearing the linked list variable thePath of elements via a call to the makeEmpty method. The statement at line 13 in figure 4.11 allocates the storage for an array of numRows+BORDER elements where each element eventually holds a pointer to the storage allocated for one of the rows of the grid G. The call (line 14, figure 4.11) to the method allocateGridRows (see subsection 4.4.3) allocates the storage for each row of the grid, passing the number of grid rows, numRows, the number of grid columns, numCols, and the grid G as parameters. Next, the call (line 15, figure 4.11) to the method initializeGrid (see subsection 4.4.3) sets each square in the grid to EMPTY, passing the number of grid rows, numRows, the number of grid columns, numCols and the grid G as

parameters. At line 16 in figure 4.11, a call to the randomInt method (see section 4.2) with parameter values numCols and 1 randomly chooses, from the range 1 to numCols, the x coordinate of the path's starting square, pathStartX. At line 17 in figure 4.11, a call to the randomInt method (see section 4.2) with parameter values numRows and 1 randomly chooses, from the range 1 to numRows, the y coordinate of the path's starting square, pathStartY. Line 18 in figure 4.11 sets the path's starting square, G[pathStartY][pathStartX], to the value SQUARE. Line 19 in figure 4.11 computes the size of the range of path length values, pathRange. At lines 20 and 21 in figure 4.11, a call to the randomInt method (see section 4.2) with parameter values pathRange and minPathLength randomly chooses, from the range minPathLength to minPathLength+pathRange, the path length value, pathLength. Line 22 in figure 4.11 assigns the variable pathLengthCount to pathLength-1 which is the remaining number of path squares to add to the grid.

```
 1: void Path::createNewCrossoverPath
 2:               (int numRows, int numCols,
 3:                int minPathLength, int maxPathLength)
 4: {
 5:   int i, xPos, yPos, pathRange, pathLengthCount;
 6:   int inc, direction, legLength;
 7:   int *G[];
 8:   int dirOffsets[NUMDIR][NUMDIM] = {{0, -1}, {1, 0},
 9:                                     {0, 1},  {-1, 0}};
10:   PathLeg pl;
11:
12:   thePath.makeEmpty();
13:   G = new int*[numRows+BORDER];
14:   allocateGridRows(numRows, numCols, G);
15:   initializeGrid(numRows, numCols, G);
16:   pathStartX = Random::randomInt(numCols, 1);
17:   pathStartY = Random::randomInt(numRows, 1);
18:   G[pathStartY][pathStartX] = SQUARE;
19:   pathRange = maxPathLength - minPathLength + 1;
20:   pathLength = Random::randomInt(pathRange,
```

```
21:                                    minPathLength);
22:   pathLengthCount = pathLength - 1;
23:   xPos = pathStartX;
24:   yPos = pathStartY;
25:   direction = chooseDirection(numRows, numCols,
26:                               xPos, yPos);
27:   inc = chooseIncrement(numRows, numCols, direction,
28:                         xPos, yPos);
29:   legLength = 0;
30:   do {
31:     for (i=1; i<=inc && pathLengthCount>0; i++)
32:     {
33:       xPos += dirOffsets[direction][XDIR];
34:       yPos += dirOffsets[direction][YDIR];
35:       G[yPos][xPos] = SQUARE;
36:       pathLengthCount--;
37:       legLength++;
38:     }
39:     pl.setPathLeg(legLength, direction);
40:     thePath.insertLast(pl);
41:     legLength = 0;
42:     chooseNewDirAndInc(direction, inc, G, xPos, yPos,
43:                        false);
44:   } while (pathLengthCount > 0);
45:
46:   chooseFSMStartPosition(numRows, numCols,
47:                          dirOffsets, G);
48:   startFSMDirection = Random::randomInt(NUMDIR, 0);
49:   deallocateGridRows(numRows, g);
50:   delete [] g;
51: }
```

Figure 4.11 - The main method to create a new crossover path.

The variables xPos and yPos hold the coordinates of the square just added to the path. Lines 23 and 24 in figure 4.11 assign xPos and yPos the coordinates of the starting square of the path, pathStartX and pathStartY, respectively. The variable *direction* contains the direction of the current leg added to the path. A call (lines 25 and 26, figure 4.11) to the method chooseDirection (see subsection 4.4.4) assigns the variable *direction* the direction of the first leg of the path, passing the number of grid rows, numRows, the

number of grid columns, numCols, and the coordinates of the starting square of the path, xPos and yPos, as parameters. A call (lines 27 and 28, figure 4.11) to the method chooseIncrement (see subsection 4.4.4) assigns the variable *inc* the maximum possible length of the first leg of the path, passing the number of grid rows, numRows, the number of grid columns, numCols, the coordinates of the starting square of the path, xPos and yPos, and the direction of the first path leg as parameters. Line 29 in figure 4.11 initializes to 0 the legLength variable containing the current leg's length.

The "do-while" loop (lines 30 to 44, figure 4.11) generates the legs of the path. The "for" loop (lines 31 to 38, figure 4.11) constructs a leg of the path one square at a time. The variable *i* is a counter variable, initialized to 1, controlling the execution of the "for" loop to terminate it (line 31, figure 4.11) when *i*'s value is greater than the variable *inc*. But, the variable pathLengthCount also controls the execution of the "for" loop possibly terminating the loop before *i*'s value is equal to *inc*. Therefore, the actual length of the current leg can be less than or equal to *inc*. Inside the body of the "for" loop at lines 33 and 34 in figure 4.11, the directional offsets in the current direction added to the coordinates (xPos, yPos) of the previous square added to the leg compute the coordinates (xPos, yPos) of the next square to add to the leg. Line 35 in figure 4.11 sets the grid square at coordinates (xPos, yPos) to the value SQUARE. Lines 36 and 37 in figure 4.11 decrement by 1 the pathLengthCount variable and increment by 1 the legLength variable to reflect the square just added to the current path leg.

The variable *pl* is an instance of the PathLeg class and stores the path leg's length and direction. After the "for" loop terminates, the call to the setPathLeg method (line 39, figure 4.11) of the PathLeg class stores the values of the legLength and direction

variables inside the PathLeg variable *pl*. A call to the insertLast method (line 40, figure 4.11) of the LinkedList class inserts a copy of the variable *pl* at the end of the LinkedList variable thePath, passing the variable *pl* as a parameter. Line 41 in figure 4.11 sets the legLength variable to 0 in order to get it ready to possibly add another leg to the path. A call (lines 42 and 43, figure 4.11) to the method chooseNewDirAndInc (see subsection 4.4.8) computes the direction and maximum possible length of the next path leg of the path, passing the direction, maximum possible length (*inc*) and coordinates (xPos, yPos) of the last square of the current path leg, the grid G and the value "false" as parameters. The direction and maximum possible length of the next path leg pass back out via the *direction* and *inc* parameters, respectively, due to pass by reference implementation. At line 44 in figure 4.11, if variable pathLengthCount is greater than 0, execute the body of the "do-while" loop (lines 30 to 44, figure 4.11) again. Otherwise, the "do-while" loop terminates.

After the "do-while" loop terminates (lines 46 and 47, figure 4.11), a call to the method chooseFSMStartPostion (see subsection 4.4.9) selects the starting position of the agent for the newly created path, passing the number of rows and columns, numRows and numCols, respectively, the directional offsets, dirOffsets, and the grid G as parameters. At line 48 in figure 4.11, a call to the randomInt method (see section 4.2) with parameter values NUMDIR and 0 randomly chooses, from the range 0 to NUMDIR-1, the starting direction of the agent, startFSMDirection. The call (line 49, figure 4.11) to the method deallocateGridRows (see subsection 4.4.3) de-allocates the storage for each row of the grid, passing the number of grid rows, numRows, and the grid G as parameters. The statement at line 50 in figure 4.11 de-allocates the storage for the array G of pointers to

the storage allocated for each row of the grid. The next subsection describes the methods to create and delete the grid.

4.4.3 The Grid Creation and Deletion Methods

The method allocateGridRows, shown in figure 4.12, allocates the storage for each row in the grid. The first two parameters of the allocateGridRows method specify the number of rows in the grid, numRows, and the number of columns in the grid, numCols. The parameter G is a one dimensional array whose size is equal to numRows+BORDER. The "for" loop at lines 6 through 9 in figure 4.12 allocates storage for a one dimensional array whose size is equal to numCols+BORDER for each element in G. In the end, G is a two-dimensional array large enough to contain the grid and a border of elements surrounding the grid.

```
 1: void Path::allocateGridRows(int numRows, int numCols,
 2:                             int G[])
 3: {
 4:   int i;
 5:
 6:   for (i=0; i<numRows+BORDER; i++)
 7:   {
 8:     G[i] = new int[numCols+BORDER];
 9:   }
10: }
```

Figure 4.12 - The method to create the grid.

The method initializeGrid in figure 4.13 initializes each element of the grid to the constant EMPTY after storage allocation of the grid. The parameters of the initializeGrid method are the number of rows in the grid, numRows, the number of columns in the grid,

numCols, and the two-dimensional array G containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns.

```
 1: void Path::initializeGrid(int numRows, int numCols,
 2:                           int *G[])
 3: {
 4:   int i, j;
 5:
 6:   for (i=0; i<numRows+BORDER; i++)
 7:   {
 8:     for (j=0; j<numCols+BORDER; j++)
 9:     {
10:       G[i][j] = EMPTY;
11:     }
12:   }
13: }
```

Figure 4.13 - The method to initialize the grid.

```
1: void Path::deallocateGridRows(int numRows, int G[])
2: {
3:   int i;
4:
5:   for (i=0; i<numRows+BORDER; i++)
6:   {
7:     delete [] (G[i]);
8:   }
9: }
```

Figure 4.14 - The method to destroy the grid.

The method deallocateGridRows in figure 4.14 performs the opposite function of the method allocateGridRows. The parameters of deallocateGridRows are the number of rows in the grid, numRows, and the one dimensional array G containing the rows of the grid. The “for” loop at lines 5 through 8 in figure 4.14 de-allocates the storage for each row of the grid stored at the corresponding element of G. The next subsection describes the methods to choose the first path leg’s direction and increment.

4.4.4 The Methods to Choose the First Path Leg's Direction and Increment

The method chooseDirection, shown in figure 4.15, chooses the direction of the first leg of the path. The first two parameters of the chooseDirection method provide the number of rows in the grid, numRows, and the number of columns in the grid, numCols. The parameters xPos and yPos hold the coordinates of the path's starting square.

```
 1: int Path::chooseDirection(int numRows, int numCols,
 2:                           int xPos, int yPos)
 3: {
 4:   int direction = Random::randomInt(NUMDIR, 0);
 5:
 6:   switch (direction) {
 7:     case NORTH: if (yPos < 3) direction = SOUTH;
 8:                 break;
 9:     case EAST:  if (xPos > (numCols-2))
10:                   direction = WEST;
11:                 break;
12:     case SOUTH: if (yPos > (numRows-2))
13:                   direction = NORTH;
14:                 break;
15:     case WEST:  if (xPos < 3) direction = EAST;
16:                 break;
17:   }
18:
19:   return direction;
20: }
```

Figure 4.15 - The method to choose the first path leg's direction.

Line 4 in figure 4.15 randomly chooses the direction of the first leg of the path as one of the values NORTH, SOUTH, EAST or WEST. But, the randomly chosen direction may not be feasible. The minimum length of a leg must be 2 because the path creation algorithm adds each new leg at the end of the current leg at a direction

perpendicular to the current leg's direction and any two legs of the path that are parallel and next to one another must have at least one EMPTY grid square between each other.

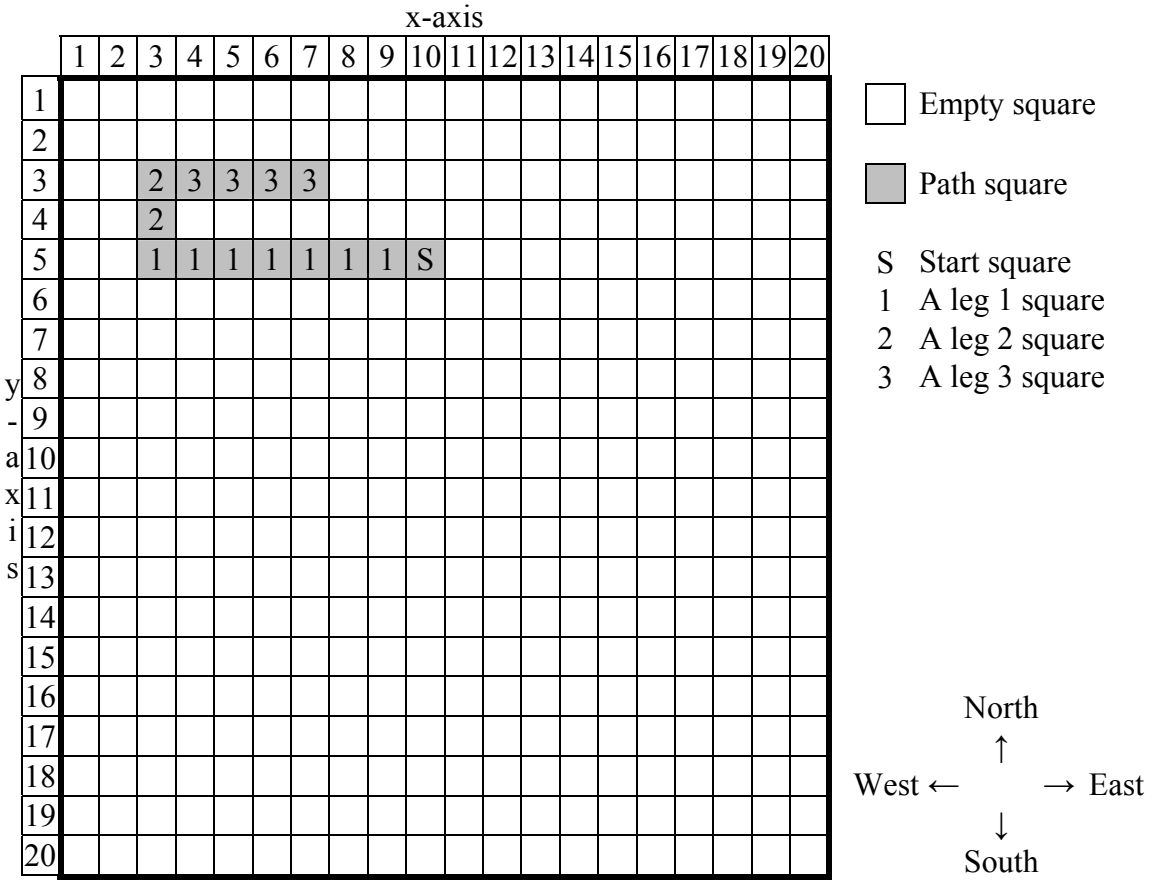


Figure 4.16 - A 20 x 20 grid containing a path.

The example in figure 4.16 illustrates this condition: S is the start path square, the squares of the first path leg contain 1, the squares of the second path leg contain 2 and the squares of the third path leg contain 3. The second leg's length of 2 enables at least one EMPTY grid square to separate the first and third legs. Hence, the "switch" statement at lines 6 through 17 in figure 4.15 performs a test to determine if the starting square of the path is less than 2 grid squares away from the edge of the grid in the randomly chosen direction. If the test is true, the "switch" statement sets the direction of the first path leg

to the opposite direction of the randomly chosen direction. For example, if the randomly chosen direction is NORTH (lines 7 and 8, figure 4.15) and the starting path square is any grid square in rows 1 or 2 (yPos < 3), the first path leg's direction changes to the opposite direction SOUTH. The method chooseDirection at line 19 in figure 4.15 returns the first path leg's direction.

```
 1: int Path::chooseIncrement(int numRows, int numCols,
 2:                           int direction,
 3:                           int xPos, int yPos)
 4: {
 5:   int inc = 0;
 6:
 7:   switch (direction) {
 8:     case NORTH: inc = Random::randomInt(yPos-2, 2);
 9:                 break;
10:     case EAST:  inc = Random::randomInt(numCols-xPos-1,
11:                                         2);
12:                 break;
13:     case SOUTH: inc = Random::randomInt(numRows-yPos-1,
14:                                         2);
15:                 break;
16:     case WEST:  inc = Random::randomInt(xPos-2, 2);
17:                 break;
18:   }
19:
20:   return inc;
21: }
```

Figure 4.17 - The method to choose the first path leg's increment.

The method chooseIncrement, shown in figure 4.17, chooses the maximum possible length of the first leg of the path. The first two parameters of the chooseIncrement method provide the number of rows in the grid, numRows, and the number of columns in the grid, numCols. The parameter *direction* contains the direction of the path's first leg. The parameters xPos and yPos hold the coordinates of the path's starting square. The "switch" statement at lines 7 through 18 in figure 4.17 provides the

mechanism to compute the maximum possible length of the first path leg according to the direction of the first path leg. The maximum possible length of the first path leg is a randomly chosen integer from the range 2 to the distance between the starting path square at coordinates (xPos, yPos) and the grid edge found in the first leg's direction. For example, assume the first leg's direction is SOUTH, the coordinates of the starting path square are (8, 10), numRows is 20 and numCols is 20. The maximum possible length of the first path length is a randomly chosen integer (lines 13 and 14, figure 4.17) from the range 2 to 10 assigned to the variable *inc*. The method chooseIncrement returns the value of *inc* (lines 20, figure 4.17) to the calling method. The next subsection describes the method to determine if the grid square ahead of the current path square is empty.

4.4.5 The Method isSquareAheadClear

The method isSquareAheadClear, shown in figure 4.18, determines if the next grid square to add to the current leg is EMPTY. The first two parameters xPos and yPos hold the coordinates of the next grid square add to the current leg. The parameter *direction* contains the direction of the path's current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. This method doesn't just check if the grid square G[yPos][xPos] is EMPTY but must check if the grid squares on either side of G[yPos][xPos] are EMPTY as well, to ensure that the newly added EMPTY grid square G[yPos][xPos] has an EMPTY grid square on both sides of it. Line 4 in figure 4.18 initializes the variable are3SquaresEmpty to true if the grid square G[yPos][xPos] is EMPTY or false otherwise. The variables eastc, westc, northc and southc (lines 5 and 6,

figure 4.18) are descriptive means to compute the coordinates for the grid squares next to grid square G[yPos][xPos] in each of the four directions. The "switch" statement, lines 8 through 19 in figure 4.18, provides the means to check the appropriate two grid squares next to the grid square G[yPos][xPos] to see if they are EMPTY. For example, according to lines 14 through 18 in figure 4.18, if the current leg's direction is EAST or WEST, the "switch" statement checks the grid squares north G[yPos][northc] and south G[yPos][southc] of grid square G[yPos][xPos]. If both grid squares are EMPTY, the variable are3SquaresEmpty remains true, otherwise are3SquaresEmpty becomes false. The method isSquareAheadClear returns the value of are3SquaresEmpty.

```
 1: bool Path::isSquareAheadClear(int xPos, int yPos,
 2:                               int direction, int *G[])
 3: {
 4:   bool are3SquaresEmpty = (G[yPos][xPos] == EMPTY);
 5:   int eastc = xPos+1, westc = xPos-1;
 6:   int northc = yPos-1, southc = yPos+1;
 7:
 8:   switch (direction) {
 9:     case NORTH:
10:     case SOUTH: are3SquaresEmpty = are3SquaresEmpty
11:                          && (G[yPos][westc] == EMPTY)
12:                          && (G[yPos][eastc] == EMPTY);
13:                 break;
14:     case EAST:
15:     case WEST:  are3SquaresEmpty = are3SquaresEmpty
16:                         && (G[northc][xPos] == EMPTY)
17:                         && (G[southc][xPos] == EMPTY);
18:                 break;
19:   }
20:
21:   return are3SquaresEmpty;
22: }
```

Figure 4.18 - The method to determine if the square ahead is empty.

The construction of the path leg 9 in figure 4.19 illustrates the use of this method. The grid square G[yPos][xPos] is G(9, 9) and the current leg's direction is EAST. Line 4 in figure 4.18 sets the variable are3SquaresEmpty to true because G[9][9] is EMPTY. Since the current leg's direction is EAST, the "switch" statement executes lines 14 through 18 in figure 4.18 examining the grid squares G[8][9] and G[10][9] to see that they are both EMPTY. Therefore, are3SquaresEmpty remains true with the method isSquareAheadClear returning are3SquaresEmpty's value. The next subsection describes the method to determine if the path creation ends.

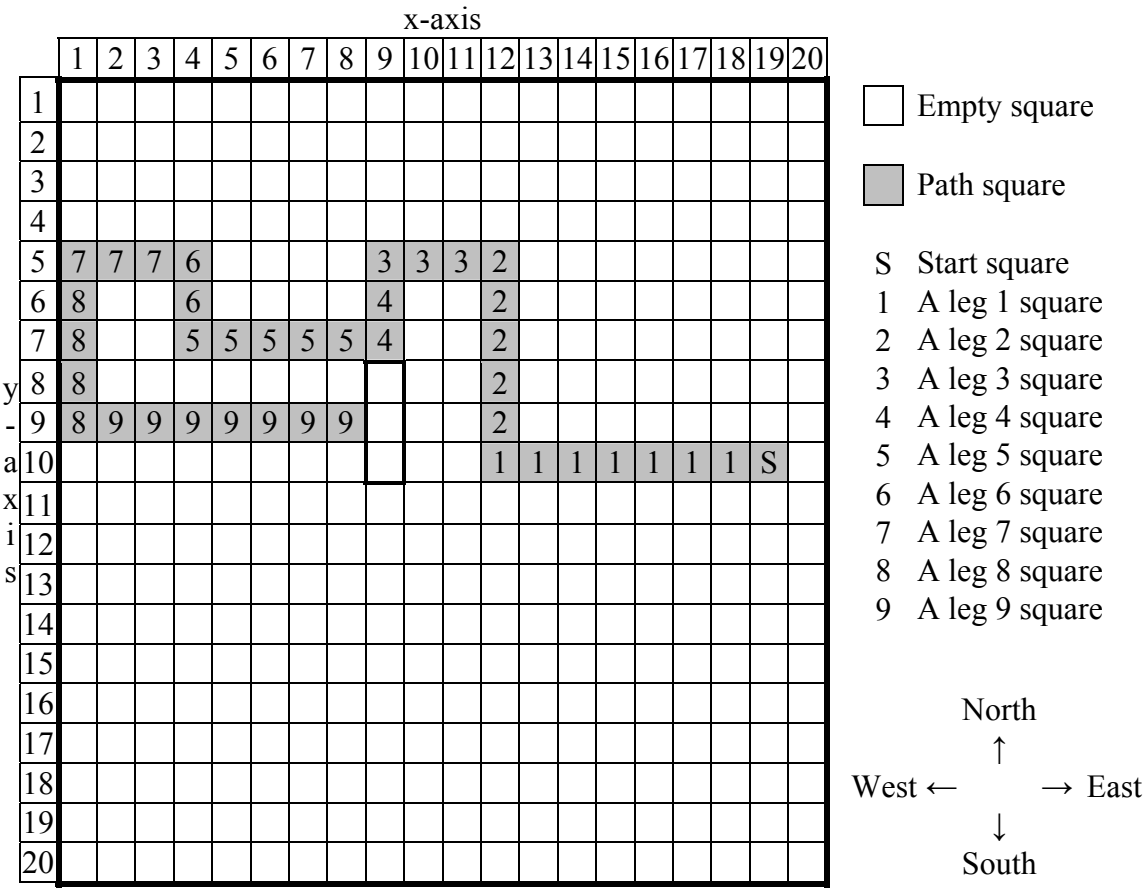


Figure 4.19 - A 20 x 20 grid containing a path.

4.4.6 The Method doesPathEnd

The method doesPathEnd, shown in figure 4.20, determines if the path creation process ends. The first two parameters xPos and yPos hold the coordinates of the last path square of the current leg. The parameter *direction* contains the direction of the path's current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The local Boolean variable *end* stores the answer to the question "does the path creation process end" with initialization to false to say that the path creation process continues. The "switch" statement at lines 6 through 15 in figure 4.20 handles the two cases: the current path leg's direction is NORTH or SOUTH, and the current path leg's direction is EAST or WEST.

```
 1: bool Path::doesPathEnd(int xPos, int yPos,
 2:                        int direction, int *G[])
 3: {
 4:   bool end = false;
 5:
 6:   switch (direction) {
 7:     case NORTH:
 8:     case SOUTH: end = !canGoEast(xPos, yPos, G) &&
 9:                       !canGoWest(xPos, yPos, G);
10:                 break;
11:     case EAST:
12:     case WEST:  end = !canGoNorth(xPos, yPos, G) &&
13:                       !canGoSouth(xPos, yPos, G);
14:                 break;
15:   }
16:
17:   return end;
18: }
```

Figure 4.20 - The method to determine if the path creation process ends.

If the current path leg's direction is NORTH or SOUTH, the path creation process can add the next path leg in the EAST or WEST direction. The methods canPathGoEast (see subsection 4.4.7) and canPathGoWest (see subsection 4.4.7) execute at lines 8 and 9 in figure 4.20. Each method returns true if the path can continue in that direction and false otherwise. If the path can not continue in both directions, assign the variable *end* the value true; otherwise, assign the variable *end* the value false.

If the current path leg's direction is EAST or WEST, the path creation process can add the next path leg in the NORTH or SOUTH direction. The methods canPathGoNorth (see subsection 4.4.7) and canPathGoSouth (see subsection 4.4.7) execute at lines 12 and 13 in figure 4.20. Each method returns true if the path can continue in that direction and false otherwise. If the path can not continue in both directions, assign the variable *end* the value true; otherwise, assign the variable *end* the value false. At line 17 in figure 4.20, the method doesPathEnd returns the value of the variable *end*. The next subsection describes all of the methods to determine if the path creation process can construct the next path leg in the appropriate directions.

4.4.7 The Methods to Determine If the Path Can Continue in Each Direction

This subsection contains the methods to determine if the path creation process can construct the next path leg in any of the four directions: NORTH, SOUTH, EAST or WEST. The first discussion is a detailed description of the methods to determine if the path creation process can create the next path leg in the NORTH direction. The main method to determine if the next path leg can go NORTH is canPathGoNorth, listed in figure 4.21. The first two parameters xPos and yPos hold the coordinates of the last

square of the current path leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The parameter noCrossover is a Boolean flag to indicate whether the path is a non-crossover or crossover path. The method canPathGoNorth returns true if the path creation process can create the next path leg in the NORTH direction and false otherwise.

```
 1: bool Path::canPathGoNorth(int xPos, int yPos,
 2                            int *G[], bool noCrossover)
 3: {
 4:   bool canGoNorth =
 5:          areCoordFarEnoughFromNorthGridEdge(yPos);
 6:
 7:   if (noCrossover)
 8:   {
 9:     canGoNorth = canGoNorth
10:       && areFirst3SquaresOfNorthLegEmpty(xPos, yPos, G)
11:       && are3SquaresWestOfNextLegEmpty(xPos, yPos, G)
12:       && are3SquaresEastOfNextLegEmpty(xPos, yPos, G);
13:   }
14:
15:   return canGoNorth;
16: }
```

Figure 4.21 - The method to determine if a leg can be added in the NORTH direction.

```
1: bool Path::areCoordFarEnoughFromNorthGridEdge(int yPos)
2: {
3:   return (yPos >= 3);
4: }
```

Figure 4.22 - The method areCoordFarEnoughFromNorthGridEdge.

The method areCoordFarEnoughFromNorthGridEdge (lines 1 to 4, figure 4.22) initializes (lines 4 and 5, figure 4.21) the variable canGoNorth by determining if the grid square G[yPos][xPos] is far enough from the NORTH edge of the grid. Recall from subsection 4.4.4, the minimum length of a leg must be 2 because the path creation

process adds each new leg to the end of the current leg at a direction perpendicular to the current leg's direction, and any two legs of the path that are parallel and next to one another must have at least one EMPTY grid square between each other. According to line 3 in figure 4.22, if the yPos coordinate places the grid square G[yPos][xPos] in any row greater than or equal to 3 then the grid square is far enough from the NORTH edge of the grid.

Certain grid squares must be EMPTY in the NORTH direction in order for the path to continue in that direction when creating a non-crossover path but not when creating a crossover path. If the noCrossover parameter is true, the path creation process creates a non-crossover path. Hence, the main method canPathGoNorth calls the method areFirst3SquaresOfNorthLegEmpty at line 10 in figure 4.21, the method are3SquaresWestOfNextLegEmpty at line 11 in figure 4.21, and the method are3SquaresEastOfNextLegEmpty at line 12 in figure 4.21. The "and" operator combines (lines 9 to 12, figure 4.21) the value of the main method variable canGoNorth and the results returned by each of these three method calls, assigning the result back to the variable canGoNorth. Figure 4.23 provides an example to illustrate the grid squares checked by the main method canPathGoNorth. Path leg 9 is the current path leg with an EAST direction and G[9][10] as the last square of path leg 9. The method areFirst3SquaresOfNorthLegEmpty tests the grid squares G[6][10], G[7][10] and G[8][10]. The method are3SquaresWestOfNextLegEmpty tests the grid squares G[6][9], G[7][9] and G[8][9]. The method are3SquaresEastOfNextLegEmpty tests the grid squares G[6][11], G[7][11] and G[8][11]. The main method canPathGoNorth returns

false because the method are3SquaresWestOfNextLegEmpty returns false, the grid squares G[6][9] and G[7][9] are not EMPTY.

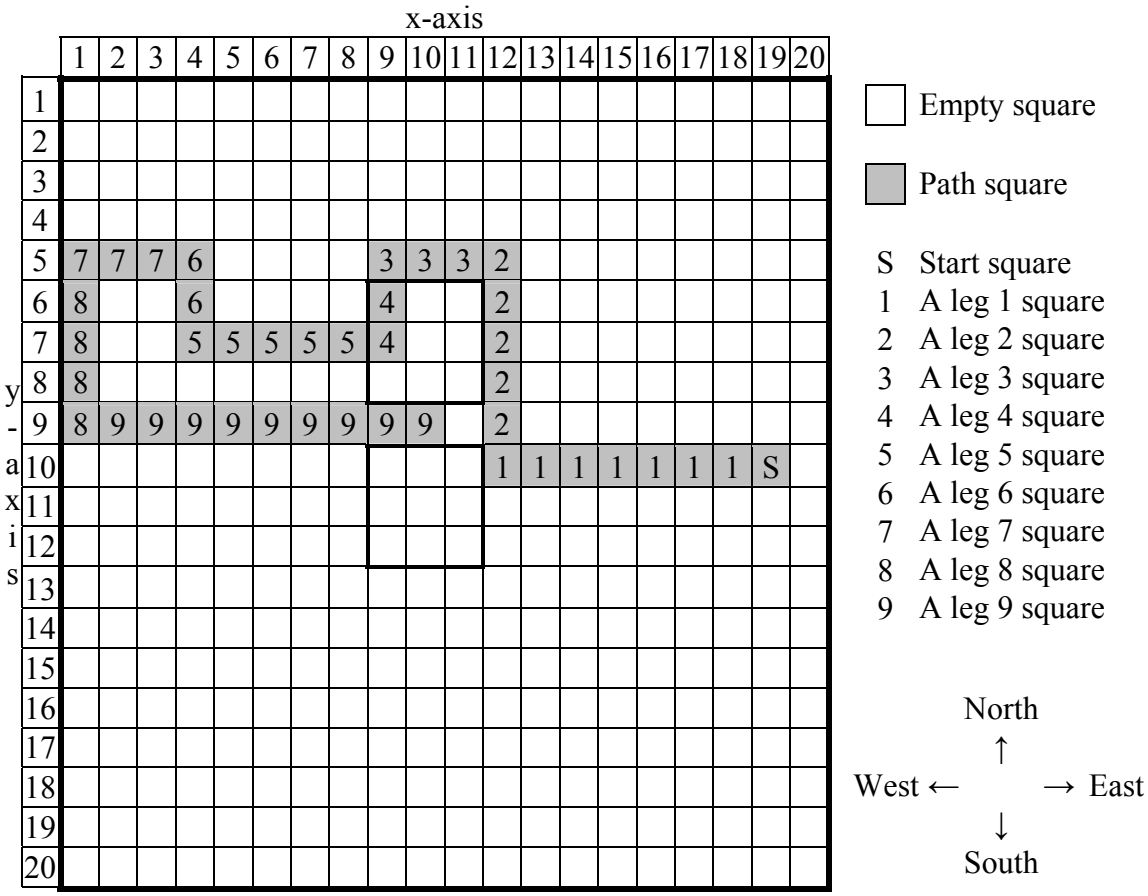


Figure 4.23 - Testing to see if the path can go north or south from leg 9.

```
 1: bool Path::areFirst3SquaresOfNorthLegEmpty
 2:                (int xPos, int yPos, int *G[])
 3: {
 4:   int north1 = yPos-1, north2 = yPos-2,
 5:       north3 = yPos-3;
 6:
 7:   return ((G[north1][xPos] == EMPTY) &&
 8:           (G[north2][xPos] == EMPTY) &&
 9:           (G[north3][xPos] == EMPTY));
10: }
```

Figure 4.24 - The method areFirst3SquaresOfNorthLegEmpty.

If the noCrossover parameter is false, the path creation process creates a crossover path leaving the variable canGoNorth's value alone. The final value of the variable canGoNorth is the value returned (line 15, figure 4.21) by the main method canPathGoNorth. The discussion of the main method canPathGoNorth continues with its three auxiliary methods.

The method areFirst3SquaresOfNorthLegEmpty (lines 1 to 10, figure 4.24) determines if the first 3 squares of the next path leg in the NORTH direction are EMPTY. This ensures that the path creation process can create the next leg in the NORTH direction with a minimal length of 2, checking the third grid square to make certain that the end of the next leg is not immediately next to an existing part of the path. The first two parameters of the method areFirst3SquaresOfNorthLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables north1, north2 and north3 (lines 4 and 5, figure 4.24) are descriptive means to compute the coordinates for the first three squares of the next path leg in the NORTH direction. The "and" operator combines (lines 7 to 9, figure 4.24) the values of the three grid squares to see if they are EMPTY with the method returning the result.

The three grid squares on both sides of the next path leg must be EMPTY to ensure that the next path leg is not next to an existing part of the path. The method are3SquaresWestOfNextLegEmpty (lines 1 to 11, figure 4.25) determines if the 3 grid squares to the WEST of the next path leg in the NORTH direction are EMPTY. The first two parameters of the method are3SquaresWestOfNextLegEmpty, xPos and yPos, hold

the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables north1, north2, north3 and westc (lines 4 to 6, figure 4.25) are descriptive means to compute the coordinates for the three grid squares to the WEST of the next path leg in the NORTH direction. The "and" operator combines (lines 8 to 10, figure 4.25) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::are3SquaresWestOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int north1 = yPos-1, north2 = yPos-2,
 5:       north3 = yPos-3;
 6:   int westc = xPos-1;
 7:
 8:   return ((G[north1][westc] == EMPTY) &&
 9:           (G[north2][westc] == EMPTY) &&
10:           (G[north3][westc] == EMPTY));
11: }
```

Figure 4.25 - The method are3SquaresWestOfNextLegEmpty.

```
 1: bool Path::are3SquaresEastOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int north1 = yPos-1, north2 = yPos-2,
 5:       north3 = yPos-3;
 6:   int eastc = xPos+1;
 7:
 8:   return ((G[north1][eastc] == EMPTY) &&
 9:           (G[north2][eastc] == EMPTY) &&
10:           (G[north3][eastc] == EMPTY));
11: }
```

Figure 4.26 - The method are3SquaresEastOfNextLegEmpty.

The method are3SquaresEastOfNextLegEmpty (lines 1 to 11, figure 4.26) determines if the 3 grid squares to the EAST of the next path leg in the NORTH direction are EMPTY. The first two parameters of the method are3SquaresEastOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables north1, north2, north3 and eastc (lines 4 to 6, figure 4.26) are descriptive means to compute the coordinates for the three grid squares to the EAST of the next path leg in the NORTH direction. The "and" operator combines (lines 8 to 10, figure 4.26) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::canPathGoSouth(int xPos, int yPos,
 2:                           int *G[], bool noCrossover)
 3: {
 4:   bool canGoSouth =
 5:          areCoordFarEnoughFromSouthGridEdge(yPos);
 6:
 7:   if (noCrossover)
 8:   {
 9:     canGoSouth = canGoSouth
10:       && areFirst3SquaresOfSouthLegEmpty(xPos, yPos, G)
11:       && are3SquaresWestOfNextLegEmpty(xPos, yPos, G)
12:       && are3SquaresEastOfNextLegEmpty(xPos, yPos, G);
13:   }
14:
15:   return canGoSouth;
16: }
```

Figure 4.27 - The method to determine if a leg can be added in the SOUTH direction.

The discussion continues with the detailed description of the methods to determine if the path creation process can create the next path leg in the SOUTH direction. The main method to determine if the next path leg can go SOUTH is

canPathGoSouth, listed in figure 4.27. The first two parameters xPos and yPos hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The parameter noCrossover is a Boolean flag to indicate whether or the path is a non-crossover path or crossover path. The method canPathGoSouth returns true if the path creation process can create the next path leg in the SOUTH direction and false otherwise.

```
1: bool Path::areCoordFarEnoughFromSouthGridEdge
2:                (int yPos, int rows)
3: {
4:   return (yPos <= (rows-2));
5: }
```

Figure 4.28 - The method areCoordFarEnoughFromSouthGridEdge.

The method areCoordFarEnoughFromSouthGridEdge (lines 1 to 5, figure 4.28) initializes (lines 4 and 5, figure 4.27) the variable canGoSouth by determining if the grid square G[yPos][xPos] is far enough from the SOUTH edge of the grid. Recall from subsection 4.4.4, the minimum length of a leg must be 2 because the path creation process adds each new leg to the end of the current leg at a direction perpendicular to the current leg's direction, and any two legs of the path that are parallel and next to one another must have at least one EMPTY grid square between each other. According to line 4 in figure 4.28, if the yPos coordinate places the grid square G[yPos][xPos] in any row less than or equal to (rows-2) then the grid square is far enough from the SOUTH edge of the grid.

Certain grid squares must be EMPTY in the SOUTH direction in order for the path to continue in that direction when creating a non-crossover path but not when

creating a crossover path. If the noCrossover parameter is true, the path creation process creates a non-crossover path. Hence, the main method canPathGoSouth calls the method areFirst3SquaresOfSouthLegEmpty at line 10 in figure 4.27, the method are3SquaresWestOfNextLegEmpty at line 11 in figure 4.27, and the method are3SquaresEastOfNextLegEmpty at line 12 in figure 4.27. The "and" operator combines (lines 9 to 12, figure 4.27) the value of the main method variable canGoSouth and the results returned by each of these three method calls, assigning the result back to the variable canGoSouth. Figure 4.23 provides an example to illustrate the grid squares checked by the main method canPathGoSouth. Path leg 9 is the current path leg with an EAST direction and G[9][10] as the last square of path leg 9. The method areFirst3SquaresOfSouthLegEmpty tests the grid squares G[10][10], G[11][10] and G[12][10]. The method are3SquaresWestOfNextLegEmpty tests the grid squares G[10][9], G[11][9] and G[12][9]. The method are3SquaresEastOfNextLegEmpty tests the grid squares G[10][11], G[11][11] and G[12][11]. The main method canPathGoSouth returns true because all of the grid squares tested are EMPTY.

If the noCrossover parameter is false, the path creation process creates a crossover path leaving the variable canGoSouth's value alone. The final value of the variable canGoSouth is the value returned (line 15, figure 4.27) by the main method canPathGoSouth. The discussion of the main method canGoSouth continues with its three auxiliary methods.

The method areFirst3SquaresOfSouthLegEmpty (lines 1 to 10, figure 4.29) determines if the first 3 squares of the next path leg in the SOUTH direction are EMPTY. This ensures that the path creation process can create the next leg in the SOUTH

direction with a minimal length of 2 checking the third grid square to make certain that the end of the next leg is not immediately next to an existing part of the path. The first two parameters of the method areFirst3SquaresOfSouthLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables south1, south2 and south3 (lines 4 and 5, figure 4.29) are descriptive means to compute the coordinates for the first three squares of the next path leg in the SOUTH direction. The "and" operator combines (lines 7 to 9, figure 4.29) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::areFirst3SquaresOfSouthLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int south1 = yPos+1, south2 = yPos+2,
 5:       south3 = yPos+3;
 6:
 7:   return ((G[south1][xPos] == EMPTY) &&
 8:           (G[south2][xPos] == EMPTY) &&
 9:           (G[south3][xPos] == EMPTY));
10: }
```

Figure 4.29 - The method areFirst3SquaresOfSouthLegEmpty.

The three grid squares on both sides of the next path leg must be EMPTY to ensure that the next path leg is not next to an existing part of the path. The method are3SquaresWestOfNextLegEmpty (lines 1 to 11, figure 4.30) determines if the 3 grid squares to the WEST of the next path leg in the SOUTH direction are EMPTY. The first two parameters of the method are3SquaresWestOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-

dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables south1, south2, south3 and westc (lines 4 to 6, figure 4.30) are descriptive means to compute the coordinates for the three grid squares to the WEST of the next path leg in the SOUTH direction. The “and” operator combines (lines 8 to 10, figure 4.30) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::are3SquaresWestOfNextLegEmpty
 2:              (int xPos, int yPos, int *G[])
 3: {
 4:   int south1 = yPos+1, south2 = yPos+2,
 5:       south3 = yPos+3;
 6:   int westc = xPos-1;
 7:
 8:   return ((G[south1][westc] == EMPTY) &&
 9:           (G[south2][westc] == EMPTY) &&
10:           (G[south3][westc] == EMPTY));
11: }
```

Figure 4.30 - The method are3SquaresWestOfNextLegEmpty.

```
 1: bool Path::are3SquaresEastOfNextLegEmpty
 2:              (int xPos, int yPos, int *G[])
 3: {
 4:   int south1 = yPos+1, south2 = yPos+2,
 5:       south3 = yPos+3;
 6:   int eastc = xPos+1;
 7:
 8:   return ((G[south1][eastc] == EMPTY) &&
 9:           (G[south2][eastc] == EMPTY) &&
10:           (G[south3][eastc] == EMPTY));
11: }
```

Figure 4.31 - The method are3SquaresEastOfNextLegEmpty.

The method are3SquaresEastOfNextLegEmpty (lines 1 to 11, figure 4.31) determines if the 3 grid squares to the EAST of the next path leg in the SOUTH direction

are EMPTY. The first two parameters of the method are3SquaresEastOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables south1, south2, south3 and eastc (lines 4 to 6, figure 4.31) are descriptive means to compute the coordinates for the three grid squares to the EAST of the next path leg in the SOUTH direction. The "and" operator combines (lines 8 to 10, figure 4.31) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::canPathGoEast(int xPos, int yPos,
 2:                          int *G[], bool noCrossover)
 3: {
 4:   bool canGoEast =
 5:          areCoordFarEnoughFromEastGridEdge(xPos);
 6:
 7:   if (noCrossover)
 8:   {
 9:     canGoEast = canGoEast
10:       && areFirst3SquaresOfEastLegEmpty(xPos, yPos, G)
11:       && are3SquaresNorthOfNextLegEmpty(xPos, yPos, G)
12:       && are3SquaresSouthOfNextLegEmpty(xPos, yPos, G);
13:   }
14:
15:   return canGoEast;
16: }
```

Figure 4.32 - The method to determine if a leg can be added in the EAST direction.

The discussion continues with the detailed description of the methods to determine if the path creation process can create the next path leg in the EAST direction. The main method to determine if the next path leg can go EAST is canPathGoEast, listed in figure 4.32. The first two parameters xPos and yPos hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing

the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The parameter noCrossover is a Boolean flag to indicate whether the path is a non-crossover or crossover path. The method canPathGoEast returns true if the path creation process can create the next path leg in the EAST direction and false otherwise.

```
1: bool Path::areCoordFarEnoughFromEastGridEdge
2:                (int xPos, int cols)
3: {
4:   return (xPos <= (cols-2));
5: }
```

Figure 4.33 - The method areCoordFarEnoughFromEastGridEdge.

The method areCoordFarEnoughFromEastGridEdge (lines 1 to 5, figure 4.33) initializes (lines 4 and 5, figure 4.32) the variable canGoEast by determining if the grid square G[yPos][xPos] is far enough from the EAST edge of the grid. Recall from subsection 4.4.4, the minimum length of a leg must be 2 because the path creation process adds each new leg to the end of the current leg at a direction perpendicular to the current leg's direction, and any two legs of the path that are parallel and next to one another must have at least one EMPTY grid square between each other. According to line 4 in figure 4.33, if the xPos coordinate places the grid square G[yPos][xPos] in any column less than or equal to (cols-2) then the grid square is far enough from the EAST edge of the grid.

Certain grid squares must be EMPTY in the EAST direction in order for the path to continue in that direction when creating a non-crossover path but not when creating a crossover path. If the noCrossover parameter is true, the path creation process creates a non-crossover path. Hence, the main method canPathGoEast calls the method areFirst3SquaresOfEastLegEmpty at line 10 in figure 4.32, the method

are3SquaresNorthOfNextLegEmpty at line 11 in figure 4.32, and the method are3SquaresSouthOfNextLegEmpty at line 12 in figure 4.32. The "and" operator combines (lines 9 to 12, figure 4.32) the value of the main method variable canGoEast and the results returned by each of these three method calls, assigning the result back to the variable canGoEast.

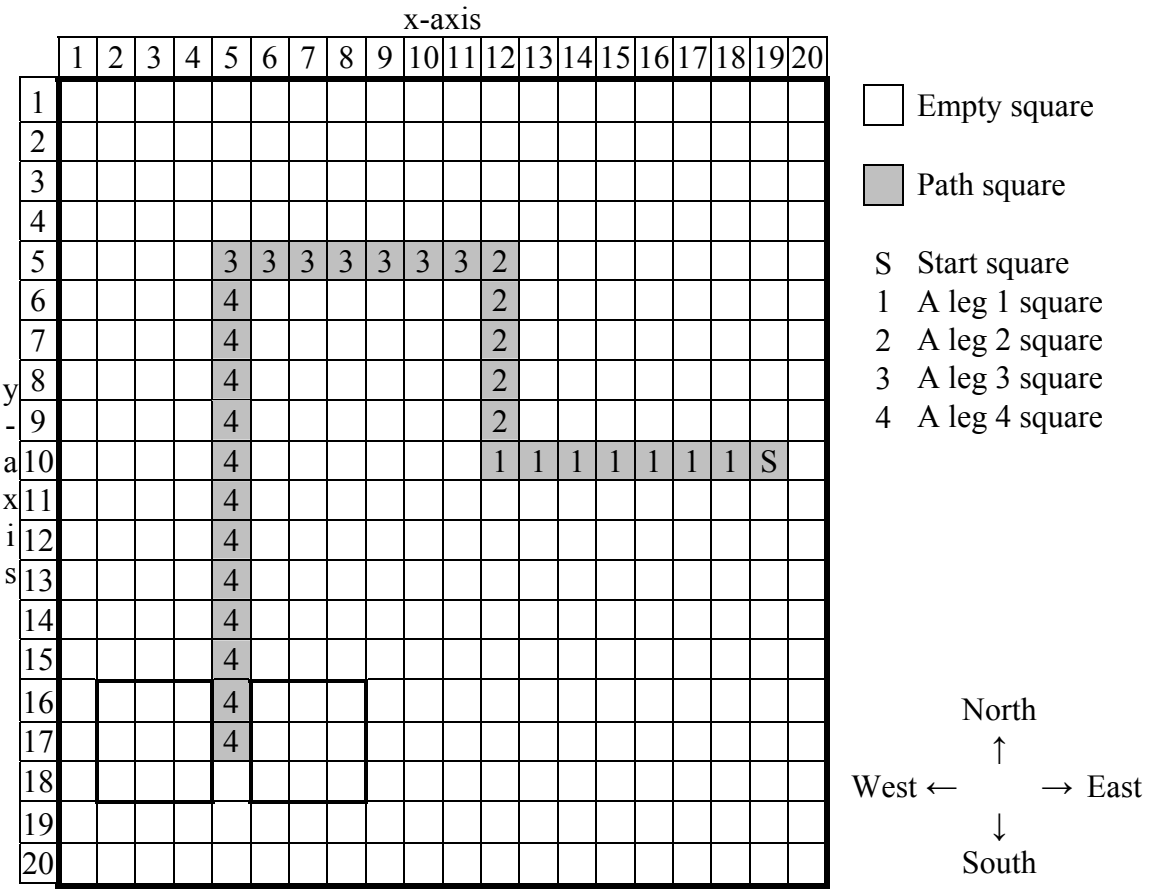


Figure 4.34 - Testing to see if the path can go west or east from leg 4.

Figure 4.34 provides an example to illustrate the grid squares checked by the main method canPathGoEast. Path leg 4 is the current path leg with a SOUTH direction and G[17][5] as the last square of path leg 4. The method areFirst3SquaresOfEastLegEmpty

tests the grid squares G[17][6], G[17][7] and G[17][8]. The method are3SquaresNorthOfNextLegEmpty tests the grid squares G[16][6], G[16][7] and G[16][8]. The method are3SquaresSouthOfNextLegEmpty tests the grid squares G[18][6], G[18][7] and G[18][8]. The main method canPathGoEast returns true because all of the grid squares tested are EMPTY.

If the noCrossover parameter is false, the path creation process creates a crossover path leaving the variable canGoEast's value alone. The final value of the variable canGoEast is the value returned (line 15, figure 4.32) by the main method canPathGoEast. The discussion of the main method canGoEast continues with its three auxiliary methods.

```
1: bool Path::areFirst3SquaresOfEastLegEmpty
2:               (int xPos, int yPos, int *G[])
3: {
4:   int east1 = xPos+1, east2 = xPos+2, east3 = xPos+3;
5:
6:   return ((G[yPos][east1] == EMPTY) &&
7:           (G[yPos][east2] == EMPTY) &&
8:           (G[yPos][east3] == EMPTY));
9: }
```

Figure 4.35 - The method areFirst3SquaresOfEastLegEmpty.

The method areFirst3SquaresOfEastLegEmpty (lines 1 to 9, figure 4.35) determines if the first 3 grid squares of the next path leg in the EAST direction are EMPTY. This ensures that the path creation process can create the next leg in the EAST direction with a minimal length of 2, checking the third grid square to make certain that the end of the next leg is not immediately next to an existing part of the path. The first two parameters of the method areFirst3SquaresOfEastLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-

dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables east1, east2 and east3 (line 4, figure 4.35) are descriptive means to compute the coordinates for the first three grid squares of the next path leg in the EAST direction. The "and" operator combines (lines 6 to 8, figure 4.35) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::are3SquaresNorthOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int east1 = xPos+1, east2 = xPos+2, east3 = xPos+3;
 5:   int northc = yPos-1;
 6:
 7:   return ((G[northc][east1] == EMPTY) &&
 8:           (G[northc][east2] == EMPTY) &&
 9:           (G[northc][east3] == EMPTY));
10: }
```

Figure 4.36 - The method are3SquaresNorthOfNextLegEmpty.

The three grid squares on both sides of the next path leg must be EMPTY to ensure that the next path leg is not next to an existing part of the path. The method are3SquaresNorthOfNextLegEmpty (lines 1 to 10, figure 4.36) determines if the 3 grid squares to the NORTH of the next path leg in the EAST direction are EMPTY. The first two parameters of the method are3SquaresNorthOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables east1, east2, east3 and northc (lines 4 and 5, figure 4.36) are descriptive means to compute the coordinates for the three grid squares to the NORTH of the next path leg in the EAST direction. The "and" operator

combines (lines 7 to 9, figure 4.36) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::are3SquaresSouthOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int east1 = xPos+1, east2 = xPos+2, east3 = xPos+3;
 5:   int southc = yPos+1;
 6:
 7:   return ((G[southc][east1] == EMPTY) &&
 8:           (G[southc][east2] == EMPTY) &&
 9:           (G[southc][east3] == EMPTY));
10: }
```

Figure 4.37 - The method are3SquaresSouthOfNextLegEmpty.

The method are3SquaresSouthOfNextLegEmpty (lines 1 to 10, figure 4.37) determines if the 3 grid squares to the SOUTH of the next path leg in the EAST direction are EMPTY. The first two parameters of the method are3SquaresSouthOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables east1, east2, east3 and southc (lines 4 and 5, figure 4.37) are descriptive means to compute the coordinates for the three grid squares to the SOUTH of the next path leg in the EAST direction. The “and” operator combines (lines 7 to 9, figure 4.37) the values of the three grid squares to see if they are EMPTY with the method returning the result.

The discussion continues with the detailed description of the methods to determine if the path creation process can create the next path leg in the WEST direction. The main method to determine if the next path leg can go WEST is canPathGoWest, listed in figure 4.38. The first two parameters xPos and yPos hold the coordinates of the

last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The parameter noCrossover is a Boolean flag to indicate whether the path is a non-crossover or crossover path. The method canPathGoWest returns true if the path creation process can create the next path leg in the WEST direction and false otherwise.

```
 1: bool Path::canPathGoWest(int xPos, int yPos,
 2:                          int *G[], bool noCrossover)
 3: {
 4:   bool canGoWest =
 5:          areCoordFarEnoughFromWestGridEdge(xPos);
 6:
 7:   if (noCrossover)
 8:   {
 9:     canGoWest = canGoWest
10:       && areFirst3SquaresOfWestLegEmpty(xPos, yPos, G)
11:       && are3SquaresNorthOfNextLegEmpty(xPos, yPos, G)
12:       && are3SquaresSouthOfNextLegEmpty(xPos, yPos, G);
13:   }
14:
15:   return canGoWest;
16: }
```

Figure 4.38 - The method to determine if a leg can be added in the WEST direction.

```
1: bool Path::areCoordFarEnoughFromWestGridEdge(int xPos)
2: {
3:   return (xPos >= 3);
4: }
```

Figure 4.39 - The method areCoordFarEnoughFromWestGridEdge.

The method areCoordFarEnoughFromWestGridEdge (lines 1 to 4, figure 4.39) initializes (lines 4 and 5, figure 4.38) the variable canGoWest by determining if the grid square G[yPos][xPos] is far enough from the WEST edge of the grid. Recall from subsection 4.4.4, the minimum length of a leg must be 2 because the path creation

process adds each new leg to the end of the current leg at a direction perpendicular to the current leg's direction, and any two legs of the path that are parallel and next to one another must have at least one EMPTY grid square between each other. According to line 3 in figure 4.39, if the xPos coordinate places the grid square G[yPos][xPos] in any column greater than or equal to 3 then the grid square is far enough from the WEST edge of the grid.

Certain grid squares must be EMPTY in the WEST direction in order for the path to continue in that direction when creating a non-crossover path but not when creating a crossover path. If the noCrossover parameter is true, the path creation process creates a non-crossover path. Hence, the main method canPathGoWest calls the method areFirst3SquaresOfWestLegEmpty at line 10 in figure 4.38, the method are3SquaresNorthOfNextLegEmpty at line 11 in figure 4.38, and the method are3SquaresSouthOfNextLegEmpty at line 12 in figure 4.38. The "and" operator combines (lines 9 to 12, figure 4.38) the value of the main method variable canGoWest and the results returned by each of these three method calls, assigning the result back to the variable canGoWest. Figure 4.34 provides an example to illustrate the grid squares checked by the main method canPathGoWest. Path leg 4 is the current path leg with a SOUTH direction and G[17][5] as the last square of path leg 4. The method areFirst3SquaresOfWestLegEmpty tests the grid squares G[17][2], G[17][3] and G[17][4]. The method are3SquaresNorthOfNextLegEmpty tests the grid squares G[16][2], G[16][3] and G[16][4]. The method are3SquaresSouthOfNextLegEmpty tests the grid squares G[18][2], G[18][3] and G[18][4]. The main method canPathGoWest returns true because all of the grid squares tested are EMPTY.

If the noCrossover parameter is false, the path creation process creates a crossover path leaving the variable canGoWest's value alone. The final value of the variable canGoWest is the value returned (line 15, figure 4.38) by the main method canPathGoWest. The remainder of this subsection describes the three auxiliary methods used by the main method canGoWest.

```
1: bool Path::areFirst3SquaresOfWestLegEmpty
2:                (int xPos, int yPos, int *G[])
3: {
4:   int west1 = xPos-1, west2 = xPos-2, west3 = xPos-3;
5:
6:   return ((G[yPos][west1] == EMPTY) &&
7:           (G[yPos][west2] == EMPTY) &&
8:           (G[yPos][west3] == EMPTY));
9: }
```

Figure 4.40 - The method areFirst3SquaresOfWestLegEmpty.

The method areFirst3SquaresOfWestLegEmpty (lines 1 to 9, figure 4.40) determines if the first 3 grid squares of the next path leg in the WEST direction are EMPTY. This ensures that the path creation process can create the next leg in the WEST direction with a minimal length of 2, checking the third grid square to make certain that the end of the next leg is not immediately next to an existing part of the path. The first two parameters of the method areFirst3SquaresOfWestLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables west1, west2 and west3 (line 4, figure 4.40) are descriptive means to compute the coordinates for the first three grid squares of the next path leg in the WEST direction. The "and" operator combines (lines 6 to 8,

figure 4.40) the values of the three grid squares to see if they are EMPTY with the method returning the result.

```
 1: bool Path::are3SquaresNorthOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int west1 = xPos-1, west2 = xPos-2, west3 = xPos-3;
 5:   int northc = yPos-1;
 6:
 7:   return ((G[northc][west1] == EMPTY) &&
 8:           (G[northc][west2] == EMPTY) &&
 9:           (G[northc][west3] == EMPTY));
10: }
```

Figure 4.41 - The method are3SquaresNorthOfNextLegEmpty.

The three grid squares on both sides of the next path leg must be EMPTY to ensure that the next path leg is not next to an existing part of the path. The method are3SquaresNorthOfNextLegEmpty (lines 1 to 10, figure 4.41) determines if the 3 grid squares to the NORTH of the next path leg in the WEST direction are EMPTY. The first two parameters of the method are3SquaresNorthOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables west1, west2, west3 and northc (lines 4 and 5, figure 4.41) are descriptive means to compute the coordinates for the three grid squares to the NORTH of the next path leg in the WEST direction. The “and” operator combines (lines 7 to 9, figure 4.41) the values of the three grid squares to see if they are EMPTY with the method returning the result.

The method are3SquaresSouthOfNextLegEmpty (lines 1 to 10, figure 4.42) determines if the 3 grid squares to the SOUTH of the next path leg in the WEST direction

are EMPTY. The first two parameters of the method are3SquaresSouthOfNextLegEmpty, xPos and yPos, hold the coordinates of the last path square of the current leg. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The variables west1, west2, west3 and southc (lines 4 and 5, figure 4.42) are descriptive means to compute the coordinates for the three grid squares to the SOUTH of the next path leg in the WEST direction. The "and" operator combines (lines 7 to 9, figure 4.42) the values of the three grid squares to see if they are EMPTY with the method returning the result. Assuming the path creation process can add a new path leg, the method in the next subsection chooses the direction and length of the next path leg.

```
 1: bool Path::are3SquaresSouthOfNextLegEmpty
 2:               (int xPos, int yPos, int *G[])
 3: {
 4:   int west1 = xPos-1, west2 = xPos-2, west3 = xPos-3;
 5:   int southc = yPos+1;
 6:
 7:   return ((G[southc][west1] == EMPTY) &&
 8:           (G[southc][west2] == EMPTY) &&
 9:           (G[southc][west3] == EMPTY));
10: }
```

Figure 4.42 - The method are3SquaresSouthOfNextLegEmpty.

4.4.8 The Method chooseNewDirAndInc

The method chooseNewDirAndInc, shown in figure 4.43, computes the direction and increment of the next path leg. The parameters *direction* and *inc* contain the direction and increment, respectively, of the current path leg and pass the direction and increment, respectively, of the next path leg out to the calling method via pass by

reference. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns. The parameters numRows and numCols provide the number of rows and columns, respectively, in the grid. The parameters xPos and yPos hold the coordinates of the last path square of the current leg. The parameter noCrossover is a Boolean flag to indicate whether the path is a non-crossover or crossover path.

```
 1: void Path::chooseNewDirAndInc(int &direction, int &inc,
 2:                               int *G[],
 3:                               int numRows, int numCols,
 4:                               int xPos, int yPos)
 5: {
 6:   int choice = Random::randomInt(2, 0);
 7:
 8:   switch (direction) {
 9:     case NORTH:
10:     case SOUTH: switch (choice) {
11:                   case 0: direction = EAST;
12:                           break;
13:                   case 1: direction = WEST;
14:                           break;
15:                 }
16:                 break;
17:     case EAST:
18:     case WEST:  switch (choice) {
19:                   case 0: direction = NORTH;
20:                           break;
21:                   case 1: direction = SOUTH;
22:                           break;
23:                 }
24:                 break;
25:   }
26:   switch (direction) {
27:     case NORTH: if (!canGoNorth(xPos, yPos, G))
28:                   direction = SOUTH;
29:                 break;
30:     case EAST:  if (!canGoEast(xPos, yPos, G))
31:                   direction = WEST;
32:                 break;
33:     case SOUTH: if (!canGoSouth(xPos, yPos, G))
34:                   direction = NORTH;
```

```
35:                   break;
36:     case WEST:  if (!canGoWest(xPos, yPos, G))
37:                   direction = SOUTH;
38:                 break;
39:   }
40:   switch (direction) {
41:     case NORTH: inc = Random::randomInt(yPos-2, 2);
42:                 break;
43:     case EAST:  inc = Random::randomInt(numCols-xPos-1,
44:                                         2);
45:                 break;
46:     case SOUTH: inc = Random::randomInt(numRows-yPos-1,
47:                                         2);
48:                 break;
49:     case WEST:  inc = Random::randomInt(xPos-2, 2);
50:                 break;
51:   }
52: }
```

Figure 4.43 - The method to choose the direction and increment of the next path leg.

The path creation process adds each new path leg at a direction perpendicular to the direction of the current path leg. The chooseNewDirAndInc method executes only when the path creation process can create the next path leg in at least one of the two possible directions. A call to the randomInt method (see section 4.2) at line 6 in figure 4.43 randomly assigns one of the two integers, 0 or 1, to the variable *choice*. The "switch" statement at lines 8 through 25 in figure 4.43 uses the value of *choice* to assign the actual direction of the next path leg to the variable *direction*. If the current path leg's direction is NORTH or SOUTH (lines 9 to 16, figure 4.43) then the next path leg's direction is either EAST or WEST. The "switch" statement at lines 10 through 15 in figure 4.43 uses the value of *choice* to assign the variable *direction*, the new direction (EAST or WEST) of the next path leg. Likewise, if the current path leg's direction is EAST or WEST (lines 17 to 24, figure 4.43) then the next path leg's direction is either NORTH or SOUTH. The "switch" statement at lines 18 through 23 in figure 4.43 uses

the value of *choice* to assign the variable *direction*, the new direction (NORTH or SOUTH) of the next path leg.

The direction chosen for the next path leg may not be valid. It is possible that the path creation process can create the next path leg in only one of the two possible directions. The "switch" statement at lines 26 through 39 in figure 4.43 ensures that the path creation process can create the next path leg in the chosen direction. If the chosen direction is not valid, then change the direction to the opposite direction. For example, if the next path leg's chosen direction is NORTH and a call to the method canPathGoNorth (lines 27 and 28, figure 4.43) returns false then set the variable *direction* to SOUTH.

Next, the "switch" statement at lines 40 through 51 in figure 4.43 provides the mechanism to compute the maximum possible length of the next path leg according to its direction. The path creation process randomly chooses the maximum possible length of the next path leg from the integer range 2 to the distance between the coordinates of the last square of the current path leg (xPos, yPos) and the grid edge found in the next path leg's direction. For example, assume the current path leg's direction is SOUTH, the coordinates of the last square of the current path leg is (8, 10), numRows is 20 and numCols is 20. The path creation process randomly chooses an integer (lines 46 and 47, figure 4.43) from the range 2 to 10 as the maximum possible length of the next path length assigning it to the parameter *inc*. Because the method chooseNewDirAndInc implements the parameters *direction* and *inc* as call by reference, the new values of *direction* and *inc* pass out to the corresponding actual parameters of the call to chooseNewDirAndInc. The next subsection describes the method to choose the agent's starting position.

4.4.9 The Method chooseFSMStartPosition

The method chooseFSMStartPosition, shown in figure 4.44, randomly chooses the agent's starting position for the newly created path. The first two parameters of the chooseFSMStartPosition method provide the number of rows in the grid, numRows, and the number of columns in the grid, numCols.

```
 1: void Path::chooseFSMStartPosition
 2:               (int numRows, int numCols,
 3:                int dirOffsets[NUMDIR][NUMDIM], int *G[])
 4: {
 5:   bool terminate;
 6:   int i, xPos, yPos, direction;
 7:
 8:   terminate = false;
 9:   direction = Random::randomInt(NUMDIR, 0);
10:   for (i=0; i<NUMDIR && !terminate; i++)
11:   {
12:     xPos = pathStartX + dirOffsets[direction][XDIR];
13:     yPos = pathStartY + dirOffsets[direction][YDIR];
14:     if ((xPos >= 1) && (xPos <= numCols) &&
15:         (yPos >= 1) && (yPos <= numRows) &&
16:         (G[yPos][xPos] == EMPTY))
17:     {
18:       terminate = true;
19:     }
20:     else
21:     {
22:       direction = (direction + 1) % NUMDIR;
23:     }
24:   }
25:   startFSMX = xPos;
26:   startFSMY = yPos;
27: }
```

Figure 4.44 - The method to choose the agent's starting position for the path.

The parameter dirOffsets is an array that stores the directional offsets needed to compute the coordinates of the grid square next to the current grid square for any of the

four directions: NORTH, EAST, SOUTH and WEST. The parameter G is the two-dimensional array containing the grid having (numRows + BORDER) rows and (numCols + BORDER) columns.

The starting position of the agent for the path is one of the four grid squares next to the path's starting square in the NORTH, EAST, SOUTH and WEST directions. A call to the randomInt method (see section 4.2) randomly chooses the initial direction for the search of the agent's starting position, assigning the direction (line 9, figure 4.44) to the variable *direction*. The variable *terminate*, initialized to false at line 8 in figure 4.44, is the flag to stop the "for" loop (lines 10 to 24, figure 4.44) when the loop encounters the first empty grid square. The counter variable *i* also controls the "for" loop, allowing the loop to examine each of the four grid squares next to the path's starting square in the NORTH, EAST, SOUTH and WEST directions. The method chooseFSMStartPosition is a method in the Path class. The variables pathStartX and pathStartY are class variables of the Path class to contain the coordinates of the path's starting square. The Path class initializes the value of these two class variables prior to the execution of this method, chooseFSMStartPosition. The path creation process computes (lines 12 and 13, figure 4.44) the x and y coordinates (xPos, yPos) of the agent's possible starting position by adding the directional offsets in the current direction to the coordinates (pathStartX, pathStartY) of the path's starting square. If the coordinates (xPos, yPos) are within the grid (lines 14 and 15, figure 4.44) and the grid square at G[yPos][xPos] is EMPTY (line 16, figure 4.44) then set the variable *terminate* to true (line 18, figure 4.44) in order to terminate the "for" loop early because the path creation process found the agent's starting position. Otherwise, set *direction* to the next direction in the sequence, see figure 4.6. At

line 22 in figure 4.44, increment *direction* by 1 and mod the result by the number of directions (NUMDIR) to ensure that the last direction, WEST, cycles back to the first direction, NORTH. Once the "for" loop terminates, assign (lines 25 and 26, figure 4.44) the Path class variables startFSMX and startFSMY the coordinates of the agent's starting position from the variables xPos and yPos, respectively. The next section discusses the fitness functions used in the various signature recognition program experiments.

4.5 The Fitness Functions

This section provides a detailed description of the methods to compute the fitness value of an agent for a particular path. The next subsection describes the main method computeFSMFitness to compute the agent's fitness value for a particular path regardless of the order of path square consumption. Subsection 4.5.2 describes the main method computeFSMPCFitness to compute the agent's fitness value for a particular path but only path square consumption in a partial contiguous order. The subsections 4.5.3 and 4.5.4 provide descriptions of the auxiliary methods used by each main method.

4.5.1 The Method computeFSMFitness

This fitness function measures the number of path squares consumed regardless of the order of path square consumption. This method uses four of Grid's class variables: thePath, numActions, maxNumMoves and theGrid. The variable thePath, an instance of the Path class, contains all the information about the current path in the grid. The variable numActions keeps track of the number of actions the agent executed during its fitness computation. The variable maxNumMoves is the practical upper limit on the

number of actions the agent executes in trying to traverse the entire trail. The variable theGrid is the grid. The return value of computeFSMFitness is the fitness value of the agent passed into the method via the *agent* parameter whose data type is the user defined FSM class.

```
 1: int Grid::computeFSMFitness(FSM &agent)
 2: {
 3:   char fsmInput, fsmOutput;
 4:   int i, xPos, yPos;
 5:   int direction, pathLength, fitness;
 6:
 7:   agent.resetState();
 8:   resetGrid();
 9:   xPos = thePath.getStartFSMX();
10:   yPos = thePath.getStartFSMY();
11:   direction = thePath.getStartFSMDirection();
12:   pathLength = thePath.getPathLength();
13:   fitness = 0;
14:   numActions = 0;
15:   for(i=1; i<=maxNumMoves && fitness<pathLength; i++)
16:   {
17:     if (theGrid[yPos][xPos].isValid())
18:     {
19:       fitness++;
20:       theGrid[yPos][xPos].advance();
21:     }
22:     fsmInput = generateInput(xPos, yPos, direction);
23:     fsmOutput = agent.makeTransition(fsmInput);
24:     performAction(xPos, yPos, direction, fsmOutput);
25:     numActions++;
26:   }
27:   return fitness;
28: }
```

Figure 4.45 - The code for the fitness function's main method computeFSMFitness.

At line 7 in figure 4.45, the fitness computation begins by resetting the current state of the agent to the start state of 0 via the resetState method of the *agent* variable's class, FSM. For each grid square's linked list, the resetGrid method at line 8 in figure

4.45 resets the linked list pointer to point to the list's head node in order to recreate the current path in the grid. The local variables xPos and yPos store the coordinates of the current position of the agent in the grid. At lines 9 and 10 in figure 4.45, the methods getStartFSMX and getStartFSMY retrieve, from thePath, the coordinates of the agent's designated starting grid square storing them in xPos and yPos, respectively. A call to the getStartFSMDirection method (line 11, figure 4.45) of thePath variable obtains the agent's designated starting direction placing it in the variable *direction* that maintains the current direction the agent faces. At line 12 in figure 4.45, the method getPathLength retrieves, from thePath, the length of the current path in the grid storing the length in the variable pathLength. Lines 13 and 14 in figure 4.45 initialize to 0 the agent's fitness value and the number of actions executed by the agent.

Inside the "for" loop (lines 15 to 26, figure 4.45), the agent repeatedly receives an input based on its surroundings, executing an action based upon the input received and consuming the path squares encountered. The body of the "for" loop (lines 17 to 25, figure 4.45) executes so long as (line 15, figure 4.45) the number of actions the agent makes is less than or equal to maxNumMoves and the fitness value is less than the path length. The fitness function examines (line 17, figure 4.45) the linked list of the current grid square, theGrid[yPos][xPos], the agent occupies. If the current grid square's linked list contains a path square not yet consumed, the fitness function increases the agent's fitness by 1 (line 19, figure 4.45) and the fitness function changes the grid square's linked list pointer to point to the next node in the linked list (line 20, figure 4.45) to reflect that the agent consumed the path square. At line 22 in figure 4.45, a call to the method generateInput (see subsection 4.5.3) acquires the next input value for the agent. The

generateInput method receives, via its parameters, the agent's current position (xPos, yPos) and direction, and returns a character value assigning it to the fsmInput variable.

States	Inputs S	Inputs E	Inputs C
0	L/23	A/5	L/25
1	L/22	R/19	A/1
2	L/5	A/14	A/18
3	R/18	R/19	A/30
4	A/18	A/28	A/21
5	R/18	L/0	R/26
6	L/4	R/24	L/1
7	A/24	L/4	R/23
8	A/26	L/4	A/28
9	L/18	A/26	L/18
10	L/1	R/0	A/15
11	A/26	R/2	R/0
12	A/24	R/14	R/23
13	R/13	R/22	L/7
14	A/19	L/23	R/26
15	R/8	R/14	R/0
16	L/5	A/21	A/1
17	A/18	A/24	R/17
18	A/18	R/17	R/30
19	R/13	L/11	L/28
20	A/3	R/17	L/14
21	R/20	A/28	R/23
22	R/22	A/17	A/0
23	A/2	R/28	R/12
24	R/12	R/23	A/7
25	L/12	L/15	L/10
26	R/4	A/21	R/14
27	L/13	L/2	R/0
28	A/16	R/7	L/12
29	A/15	L/6	L/31
30	A/7	R/22	R/11
31	L/2	A/0	R/10

Figure 4.46 - An example of an agent.

The method makeTransition is a part of the FSM class, the data type of the agent variable. Given the input value in fsmInput and the current state of the agent, the makeTransition method retrieves, from the agent, the output and nextState values for the array element located at the (row, column) designated by (state, input). For example, if the input is "S" and the current state is 12, the makeTransition method retrieves output "A" and next state 24 from the agent shown in figure 4.46. The fitness function sets the current state of the agent to the next state value. The makeTransition method returns the output which specifies the next action the agent performs. At line 23 in figure 4.45, the makeTransition method returns the character value to assign to the fsmOutput variable.

At line 24 in figure 4.45, a call to the method performAction (see subsection 4.5.4) executes the agent's next action. The performAction method receives, via its parameters, the agent's current position (xPos, yPos), current direction and the output value in the fsmOutput variable. After the agent performs its action, the fitness function increases (line 25, figure 4.45) the agent's number of actions by 1. When the "for" loop terminates, the computeFSMFitness method returns (line 27, figure 4.45) the fitness value of the agent for the current path in the grid. The next subsection describes the second fitness function designed for the signature recognition program.

4.5.2 The Method computeFSMPCFitness

This fitness function measures the number of path squares consumed but only those path squares consumed in a partial contiguous order. This method uses five of Grid's class variables: thePath, numActions, kSegmentLength, maxNumMoves and theGrid. The variable thePath, an instance of the Path class, contains all the information

about the current path in the grid. The variable numActions keeps track of the number of actions the agent executed during its fitness computation. The variable kSegmentLength stores the length of the largest contiguous segment of the path consumed by the agent during its fitness calculation. The variable maxNumMoves is the practical upper limit on the number of actions the agent executes in trying to traverse the entire trail. The variable theGrid is the grid. The return value of computeFSMPCFitness is the fitness value of the agent passed into the method via the *agent* parameter whose data type is the user defined FSM class.

```
 1: int Grid::computeFSMPCFitness(FSM &agent)
 2: {
 3:   char fsmInput, fsmOutput;
 4:   int i, xPos, yPos, direction;
 5:   int pathLength, fitness, square;
 6:   int prevSquareConsumed, segmentLength;
 7:
 8:   agent.resetState();
 9:   resetGrid();
10:   xPos = thePath.getStartFSMX();
11:   yPos = thePath.getStartFSMY();
12:   direction = thePath.getStartFSMDirection();
13:   pathLength = thePath.getPathLength();
14:   fitness = 0;
15:   numActions = 0;
16:   prevSquareConsumed = -1;
17:   kSegmentLength = segmentLength = 0;
18:   for(i=1; i<=maxNumMoves && fitness<pathLength; i++)
19:   {
20:     if (theGrid[yPos][xPos].isValid())
21:     {
22:       square = theGrid[yPos][xPos].retrieve();
23:       theGrid[yPos][xPos].advance();
24:       if (square > prevSquareConsumed)
25:       {
26:         prevSquareConsumed = square;
27:         fitness++;
28:         if (square != prevSquareConsumed+1)
29:         {
30:           if (segmentLength > kSegmentLength)
```

```
31:            {
32:              kSegmentLength = segmentLength;
33:            }
34:            segmentLength = 0;
35:          }
36:          segmentLength++;
37:        }
38:      }
39:      fsmInput = generateInput(xPos, yPos, direction);
40:      fsmOutput = agent.makeTransition(fsmInput);
41:      performAction(xPos, yPos, direction, fsmOutput);
42:      numActions++;
43:    }
44:    return fitness;
45: }
```

Figure 4.47 - The code for the fitness function's main method computeFSMPCFitness.

At line 8 in figure 4.47, the fitness computation begins by resetting the current state of the agent to the start state of 0 via the resetState method of the *agent* variable's class, FSM. For each grid square's linked list, the resetGrid method at line 9 in figure 4.47 resets the linked list pointer to point to the list's head node in order to recreate the current path in the grid. The local variables xPos and yPos store the coordinates of the current position of the agent in the grid. At lines 10 and 11 in figure 4.47, the methods getStartFSMX and getStartFSMY retrieve, from thePath, the coordinates of the agent's designated starting grid square storing them in xPos and yPos, respectively. A call to the getStartFSMDirection method (line 12, figure 4.47) of thePath variable obtains the agent's designated starting direction placing it in the variable *direction* that maintains the current direction the agent faces. At line 13 in figure 4.47, the method getPathLength retrieves, from thePath, the length of the current path in the grid storing the length in the variable pathLength. Lines 14 and 15 in figure 4.47 initialize to 0 the agent's fitness value and the number of actions executed by the agent. The variable

prevSquareConsumed, initialized to -1 at line 16 in figure 4.47, keeps track of the integer of the last path square consumed by the agent. Line 17 in figure 4.47 initializes to 0 the variables kSegmentLength and segmentLength. The variable segmentLength stores the length of the current contiguous segment of the path consumed by the agent during its fitness calculation.

Inside the "for" loop (lines 18 to 43, figure 4.47), the agent repeatedly receives an input based on its surroundings, executing an action based upon the input received and consuming the path squares encountered. The body of the "for" loop (lines 20 to 42, figure 4.47) executes so long as (line 18, figure 4.47) the number of actions the agent makes is less than or equal to maxNumMoves and the fitness value is less than the path length. The fitness function examines (line 20, figure 4.47) the linked list of the current grid square, theGrid[yPos][xPos], the agent occupies. If the current grid square's linked list contains a path square not yet consumed, the fitness function determines (lines 22 to 37, figure 4.47) if that path square contributes to the agent's fitness value. At line 22 in figure 4.47, the fitness function retrieves, from the current grid square's linked list, the integer of the path square not yet consumed, assigning the integer to the variable *square*. At line 23 in figure 4.47, the fitness function changes the grid square's linked list pointer to point to the next node in the linked list to reflect that the agent consumed the path square. The fitness function compares the value of the current path square to the value of the previous path square consumed. If the current path square is greater than the value of the previous path square consumed (line 24, figure 4.47) then the agent consumed the current path square in a partial contiguous order and the current path square contributes (lines 26 to 36, figure 4.47) to the agent's fitness value. The order is partial because any

path squares not consumed but whose value is less than the previous path square consumed do not contribute to the agent's fitness value even if consumed later. The fitness function assigns the variable prevSquareConsumed the value of the current path square (line 26, figure 4.47) and (line 27, figure 4.47) increases the agent's fitness value by 1. Lines 28 through 36 in figure 4.47 keep track of the largest contiguous segment of the path and the current contiguous segment of the path consumed by the agent. If the current path square consumed is not the next path square after the previous path square consumed (line 27, figure 4.47) then the agent finished the current contiguous segment. Lines 30 through 34 in figure 4.47 update the value of segmentLength and possibly kSegmentLength. If the current contiguous segment length is longer than the existing largest contiguous segment length (line 30, figure 4.47) then the statement at line 32 in figure 4.47 assigns the variable kSegmentLength the value of segmentLength. The statement at line 34 in figure 4.47 resets segmentLength's value to zero to keep track of the length of the next contiguous segment of the path. After the "if" statement at lines 28 through 35 in figure 4.47, the fitness function increases the segment's length by 1.

At line 39 in figure 4.47, a call to the method generateInput (see subsection 4.5.3) acquires the next input value for the agent. The generateInput method receives, via its parameters, the agent's current position (xPos, yPos) and direction, and returns a character value assigning it to the fsmInput variable.

The method makeTransition is a part of the FSM class, the data type of the agent variable. Given the input value in fsmInput and the current state of the agent, the makeTransition method retrieves, from the agent, the output and nextState values for the array element located at the (row, column) designated by (state, input). For example, if

the input is “S” and the current state is 12, the makeTransition method retrieves output “A” and next state 24 from the agent shown in figure 4.46. The fitness function sets the current state of the agent to the next state value. The makeTransition method returns the output which specifies the next action the agent performs. At line 40 in figure 4.47, the makeTransition method returns the character value to assign to the fsmOutput variable.

At line 41 in figure 4.47, a call to the method performAction (see subsection 4.5.4) executes the agent’s next action. The performAction method receives, via its parameters, the agent’s current position (xPos, yPos), current direction and the output value in the fsmOutput variable. After the agent performs its action, the fitness function increases (line 42, figure 4.47) the agent’s number of actions by 1. When the “for” loop terminates, the computeFSMPCFitness method returns (line 44, figure 4.47) the fitness value of the agent for the current path in the grid. The next subsection describes the first of the two auxiliary methods used by the fitness functions.

4.5.3 The Method generateInput

This method uses two of Grid’s class variables: dirOffsets and theGrid. The variable dirOffsets is an array that stores the directional offsets needed to compute the coordinates of the grid square next to the current grid square for any of the four directions: NORTH, EAST, SOUTH and WEST. The variable theGrid is the grid. The generateInput method returns the input value based upon the agent’s current coordinates (xPos, yPos) and direction, passed as parameters into the generateInput method. Lines 10 and 11 in figure 4.48 compute the coordinates (xAhead, yAhead) of the grid square

immediately in front of the agent by adding the offsets of the current direction the agent faces to the agent's current coordinates (xPos, yPos).

```
 1: #define XDIM 0
 2: #define YDIM 1
 3:
 4: char Grid::generateInput(int xPos, int yPos,
 5:                          int direction)
 6: {
 7:   char input = 'E';
 8:   int xAhead, yAhead;
 9:
10:   xAhead = xPos + dirOffsets[direction][XDIM];
11:   yAhead = yPos + dirOffsets[direction][YDIM];
12:   if ((xAhead < 1) || (xAhead > numCols) ||
13:       (yAhead < 1) || (yAhead > numRows))
14:   {
15:     input = 'C';
16:   }
17:   else
18:   {
19:     if (theGrid[yAhead][xAhead].isValid())
20:     {
21:       input = 'S';
22:     }
23:     else
24:     {
25:       input = 'E';
26:     }
27:   }
28:   return input;
29: }
```

Figure 4.48 - The code for the fitness function's method to generate the agent's input.

The variable *input* holds the input value. According to lines 12 and 13 in figure 4.48, if the coordinates (xAhead, yAhead) are outside the grid, the agent faces a grid cliff. Therefore, the generateInput method (line 15, figure 4.48) assigns the value of 'C' to the variable *input*. Otherwise, if the linked list of the grid square in front of the agent, theGrid[yPos][xPos], contains a path square not yet consumed (line 19, figure 4.48) then

the generateInput method assigns the value of 'S' (line 21, figure 4.48) to the variable *input*. Otherwise, the grid square in front of the agent, theGrid[yPos][xPos], is empty and the generateInput method assigns the value of 'E' (line 25, figure 4.48) to the variable *input*. The generateInput method returns the input value (line 28, figure 4.48) back to the calling method. The next subsection describes the second auxiliary method used by the fitness functions.

4.5.4 The Method performAction

This method uses three of Grid's class variables: dirOffsets, goLeft and goRight. The variable dirOffsets is an array that stores the directional offsets needed to compute the coordinates of the grid square next to the current grid square for any of the four directions: NORTH, EAST, SOUTH and WEST. The variable goLeft is an array that stores, for each of the four directions, the new direction the agent faces after turning left by rotating 90 degrees. The variable goRight is an array that stores, for each of the four directions, the new direction the agent faces after turning right by rotating -90 degrees. The agent's current coordinates (xPos, yPos), direction and output are parameters of the performAction method. The performAction method implements the xPos, yPos and *direction* formal parameters as pass by reference to pass the values of the formal parameters back out to the corresponding actual parameters.

The value of *output* specifies the action the agent now undertakes. The "switch" statement in lines 10 through 16 in figure 4.49 handles changing the direction of the agent for each of the three possible outputs. In the case of an output value 'A', the agent does not change its direction. In the case of an output value 'L', the agent changes direction,

turning left by rotating 90 degrees. In the case of an output value 'R', the agent changes direction, turning right by rotating –90 degrees.

```
 1: #define XDIM 0
 2: #define YDIM 1
 3:
 4: void Grid::performAction(int &xPos, int &yPos,
 5:                          int &direction,
 6:                          char fsmOutput)
 7: {
 8:   int xAhead, yAhead;
 9:
10:   switch (fsmOutput) {
11:     case 'A': break;
12:     case 'L': direction = goLeft[direction];
13:               break;
14:     case 'R': direction = goRight[direction];
15:               break;
16:   }
17:   xAhead = xPos + dirOffsets[direction][XDIM];
18:   yAhead = yPos + dirOffsets[direction][YDIM];
19:   if ((xAhead >= 1) && (xAhead <= numCols) &&
20:       (yAhead >= 1) && (yAhead <= numRows))
21:   {
22:     xPos = xAhead;
23:     yPos = yAhead;
24:   }
25: }
```

Figure 4.49 - The code for the fitness function's method to perform an agent's action.

Lines 17 and 18 in figure 4.49 compute the coordinates (xAhead, yAhead) of the grid square immediately in front of the agent by adding the offsets of the current direction the agent faces to the agent's current coordinates (xPos, yPos). According to lines 19 and 10 in figure 4.49, if the coordinates (xAhead, yAhead) are inside the grid, the agent can advance forward to the grid square at the (xAhead, yAhead) coordinates. Therefore, the performAction method updates (lines 22 and 23, figure 4.49) the coordinates of the agent's current position (xPos, yPos) with the value of the (xAhead, yAhead) coordinates,

respectively. The next section provides a detailed description of the methods to create the crossover points.

4.6 Creation of the Crossover Points

Recall from section 4.3, a one-dimensional array of integers actually implements the agent table with the number of elements in the array equal to numStates. But, the agent access methods getOutput, getNextState, setOutput and setNextState allow the signature recognition program to perceive the agent as a (rectangular) two-dimensional array of numStates (32) rows and numInputs (3) columns. For the signature recognition program's genetic algorithm, a crossover point is a rectangular sub-area of the agent. This section describes the methods to create the crossover points for the signature recognition program's genetic algorithm. Figure 4.50 defines the constants used in any of the methods defined in this section. The next subsection describes the main method of the crossover point creation process. The subsections 4.6.2 through 4.6.6 provide the descriptions of the auxiliary methods used by the main method.

```
#define NUMCORNERS 4

#define COLS_DIFF 0
#define ROWS_DIFF 1

#define TOP_LFT 0
#define TOP_RGT 1
#define BOT_LFT 2
#define BOT_RGT 3
```

Figure 4.50 - The constants used by the methods defined in section 4.6.

4.6.1 The Main Method crossoverPartitions

The method crossoverPartitions is the main method called to create the crossover points for the signature recognition program's genetic algorithm through a random process to partition an agent's rectangular area into rectangular sub-areas. The parameters numRows and numCols provide the number of rows and columns, respectively, of an agent. The *partitions* parameter is an instance of the LinkedList class template in which the data type of the linked list elements is the CrossoverArea class. Each instance of the CrossoverArea class holds the length and width of a rectangular area within an agent as well as the coordinates of the top left corner of the rectangular area within an agent. Each CrossoverArea element in the *partitions* linked list is one of the crossover points of the signature recognition program's genetic algorithm.

The variable mainArea is an instance of the CrossoverArea class and stores the current rectangular sub-area processed by this method. The variable sortedPartitions is an instance of the LinkedList class template in which the data type of the linked list elements is the CrossoverArea class. The CrossoverArea elements in sortedPartitions are the unprocessed rectangular sub-areas stored in sorted order by area with the largest area at the head of the linked list. The initial rectangular area to process is the entire area of an agent whose length is numRows, width is numCols and coordinates of the top left corner are (0, 0). The call to the setCrossoverArea method (line 14, figure 4.51) of the CrossoverArea class stores inside the CrossoverArea variable mainArea the values defining the initial rectangular area: numRows, numCols, 0 and 0. A call to the insertSorted method (line 15, figure 4.51) of the LinkedList class, passing the variable

mainArea as a parameter, inserts a copy of the variable mainArea into the LinkedList variable sortedPartitions.

```
 1: void FSM::crossoverPartitions
 2:              (int numRows, int numCols,
 3:               LinkedList<CrossoverArea> &partitions)
 4: {
 5:   int corner, choice;
 6:   int rows, cols, xPos, yPos;
 7:   int newRows, newCols, newXPos, newYPos;
 8:   int newRowsDiff, newColsDiff;
 9:   int area1Rows, area1Cols, area2Rows, area2Cols;
10:   int area1XPos, area1YPos, area2XPos, area2YPos;
11:   CrossoverArea mainArea;
12:   LinkedList<CrossoverArea> sortedPartitions;
13:
14:   mainArea.setCrossoverArea(numRows, numCols, 0, 0);
15:   sortedPartitions.insertSorted(mainArea);
16:   while (!(sortedPartitions.isEmpty()))
17:   {
18:     mainArea = sortedPartitions.removeFirst();
19:     rows = mainArea.getRows();
20:     cols = mainArea.getCols();
21:     if ((rows*cols) == 1)
22:     {
23:       partitions.insertLast(mainArea);
24:     }
25:     else
26:     {
27:       xPos = mainArea.getXPos();
28:       yPos = mainArea.getYPos();
29:       newRows = Random::randomInt(rows, 1);
30:       newCols = Random::randomInt(cols, 1);
31:       corner = Random::randomInt(NUMCORNERS, 0);
32:       newRowsDiff = rows - newRows;
33:       newColsDiff = cols - newCols;
34:
35:       determineSubAreas(rows, cols, newRows, newCols,
36:                         newRowsDiff, newColsDiff,
37:                         choice, area1Rows, area1Cols,
38:                         area2Rows, area2Cols);
39:       computeArea1Coord(corner, choice, xPos, yPos,
40:                         newRowsDiff, newCols,
41:                         area1XPos, area1YPos);
42:       computeArea2Coord(corner, choice, xPos, yPos,
```

```
43:                          newRows, newColsDiff,
44:                          area2XPos, area2YPos);
45:       insertSubArea(area1Rows, area1Cols, area1XPos,
46:                     area1YPos, sortedPartitions);
47:       insertSubArea(area2Rows, area2Cols, area2XPos,
48:                     area2YPos, sortedPartitions);
49:
50:       computeNewCoord(corner, xPos, yPos, newRowsDiff,
51:                       newColsDiff, newXPos, newYPos);
52:       mainArea.setCrossoverArea(newRows, newCols,
53:                                 newXPos, newYPos);
54:       partitions.insertLast(mainArea);
55:     }
56:   }
57: }
```

Figure 4.51 - The main method to create the agent's crossover points.

The "while" loop (lines 16 to 56, figure 4.51) executes its body until the linked list sortedPartitions is empty. The method isEmpty (line 16, figure 4.51) from the LinkedList class returns true if the instance of the LinkedList class, sortedPartitions, is empty and false otherwise. At line 18 in figure 4.51, a call to the LinkedList class method removeFirst removes the CrossoverArea element at the head of the linked list sortedPartitions assigning it to the variable mainArea. At line 19 in figure 4.51, a call to the CrossoverArea method getRows retrieves the number of rows of mainArea assigning it to the variable *rows*. At line 20 in figure 4.51, a call to the CrossoverArea method getCols retrieves the number of columns of mainArea assigning it to the variable cols. If the area of mainArea is equal to 1 (line 21, figure 4.51) then mainArea automatically becomes a crossover point and a call to the insertLast method (line 23, figure 4.51) of the LinkedList class, passing mainArea as a parameter, inserts a copy of mainArea into the LinkedList variable partitions.

Otherwise, the area of mainArea is greater than 1 and the else clause (lines 25 to 55, figure 4.51) produces another crossover point from mainArea. At lines 27 and 28 in figure 4.51, a call to the CrossoverArea methods getXPos and getYPos retrieves the coordinates of the top left corner of mainArea assigning it to the variables xPos and yPos, respectively. At line 29 in figure 4.51, a call to the randomInt method (see section 4.2) with parameter values numRows and 1 randomly chooses, from the range 1 to the number of rows (numRows) of mainArea, the number of rows of the new crossover point, newRows. At line 30 in figure 4.51, a call to the randomInt method (see section 4.2) with parameter values numCols and 1 randomly chooses, from the range 1 to the number of columns (numCols) of mainArea, the number of columns of the new crossover point, newCols. A randomly chosen corner of the rectangle mainArea anchors the new crossover point within mainArea. A call to the randomInt method (line 31, figure 4.51) passing NUMCORNERS and 0 as parameters randomly chooses the corner assigning it to the variable *corner*. At line 32 in figure 4.51, the variable newRowsDiff is the difference between the number of rows in mainArea, *rows*, and the number of rows in the new crossover point, newRows. At line 33 in figure 4.51, the variable newColsDiff is the difference between the number of columns in mainArea, cols, and the number of columns in the new crossover point, newCols.

Since the new crossover point occupies a portion or the entire area of mainArea, the remaining area of mainArea can partition into 0, 1 or 2 rectangular sub-areas. Assuming the remaining area of mainArea can partition into 2 sub-areas simplifies the process for dealing with the remaining area of mainArea. A call (lines 35 to 38, figure 4.51) to the method determineSubAreas (see subsection 4.6.2) computes the number of

rows, area1Rows, and number of columns, area1Cols, for sub-area 1 and the number of rows, area2Rows, and number of columns, area2Cols, for sub-area 2, passing *rows*, cols, newRows, newCols, newRowsDiff, newColsDiff, *choice*, area1Rows, area1Cols, area2Rows and area2Cols as parameters. The method determineSubAreas implements the parameters area1Rows, area1Cols, area2Rows and area2Cols via pass by reference to pass their values back out to the calling method. A call (lines 39 and 41, figure 4.51) to the method computeArea1Coord (see subsection 4.6.3) calculates the coordinates of the top left corner of sub-area 1, passing *corner*, *choice*, xPos, yPos, newRowsDiff, newCols, area1XPos and area1YPos as parameters with the coordinates passed back out by means of the pass by reference parameters area1XPos and area1YPos. A call (lines 42 and 44, figure 4.51) to the method computeArea2Coord (see subsection 4.6.4) calculates the coordinates of the top left corner of sub-area 2, passing *corner*, *choice*, xPos, yPos, newRows, newColsDiff, area2XPos and area2YPos as parameters with the coordinates passed back out by means of the pass by reference parameters area2XPos and area2YPos. A call (lines 45 and 46, figure 4.51) to the method insertSubArea (see subsection 4.6.5) inserts sub-area 1 into the sortedPartitions linked list, passing area1Rows, area1Cols, area1XPos, area1YPos and sortedPartitions as parameters. A call (lines 47 and 48, figure 4.51) to the method insertSubArea (see subsection 4.6.5) inserts sub-area 2 into the sortedPartitions linked list, passing area2Rows, area2Cols, area2XPos, area2YPos and sortedPartitions as parameters.

A call (lines 50 and 51, figure 4.51) to the method computeNewCoord (see subsection 4.6.6) calculates the coordinates of the top left corner of the new crossover point, passing *corner*, xPos, yPos, newRowsDiff, newColsDiff, newXPos and newYPos

as parameters with the coordinates passed back out by means of the pass by reference parameters newXPos and newYPos. The call to the setCrossoverArea method (lines 52 and 53, figure 4.51) of the CrossoverArea class stores inside the CrossoverArea variable mainArea the values defining the new crossover point: newRows, newCols, newXPos and newYPos. A call to the insertLast method (line 54, figure 4.51) of the LinkedList class, passing mainArea as a parameter, inserts a copy of the new crossover point in the variable mainArea into the LinkedList variable *partitions*. The next subsection describes the first of the five auxiliary methods used by the main method crossoverPartitions.

4.6.2 The Method determineSubAreas

The method determineSubAreas, shown in figure 4.52, decides how to divide the remaining area of a rectangular portion of the agent into two sub-areas. Figure 4.53a illustrates an example of a rectangular portion of an agent, referred to here by the name mainArea, with 10 rows and 3 columns. The first two parameters of this method provide the number of rows, *rows*, and the number of columns, cols, of mainArea. In figure 4.53a, the area with 5 rows and 2 columns enclosed in a bold dashed border inside of mainArea is the new crossover point. The two parameters newRows and newCols provide the number of rows and the number of columns, respectively, of the new crossover point created inside mainArea. The parameter newRowsDiff is the difference between *rows* and newRows. The parameter newColsDiff is the difference between cols and newCols. For the example in figure 4.53a, newRowsDiff is 5 and newColsDiff is 1.

```
 1: void FSM::determineSubAreas
 2:              (int rows, int cols, int newRows,
 3:               int newCols, int newRowsDiff,
 4:               int newColsDiff, int &choice,
 5:               int &area1Rows, int &area1Cols,
 6:               int &area2Rows, int &area2Cols)
 7: {
 8:   if ((rows*newColsDiff) < (newRowsDiff*cols))
 9:   {
10:     choice = COLS_DIFF;
11:     area1Rows = rows;
12:     area1Cols = newColsDiff;
13:     area2Rows = newRowsDiff;
14:     area2Cols = newCols;
15:   }
16:   else
17:   {
18:     choice = ROWS_DIFF;
19:     area1Rows = newRows;
20:     area1Cols = newColsDiff;
21:     area2Rows = newRowsDiff;
22:     area2Cols = cols;
23:   }
24: }
```

Figure 4.52 - The method to determine the dimensions of the 2 sub-areas.

There are two different ways to divide the remaining area of mainArea into two smaller sub-areas. The first way, called COLS_DIFF, divides the remaining area of mainArea into area 1 whose dimensions are rows×newColsDiff and area 2 whose dimensions are newRowsDiff×newCols. Figure 4.53b illustrates COLS_DIFF's division of the remaining area of mainArea. Area 1 in figure 4.53b, the left sub-area enclosed in a bold dashed border, has dimensions 10×1. Area 2 in figure 4.53b, the right sub-area enclosed in a bold dashed border, has dimensions 5×2. The second way, called ROWS_DIFF, divides the remaining area of mainArea into area 1 whose dimensions are newRows×newColsDiff and area 2 whose dimensions are newRowsDiff×cols. Figure 4.53c illustrates ROWS_DIFF's division of the remaining area of mainArea. Area 1 in

figure 4.53c, the bottom sub-area enclosed in a bold dashed border, has dimensions 5×1. Area 2 in figure 4.53c, the top sub-area enclosed in a bold dashed border, has dimensions 5×3.

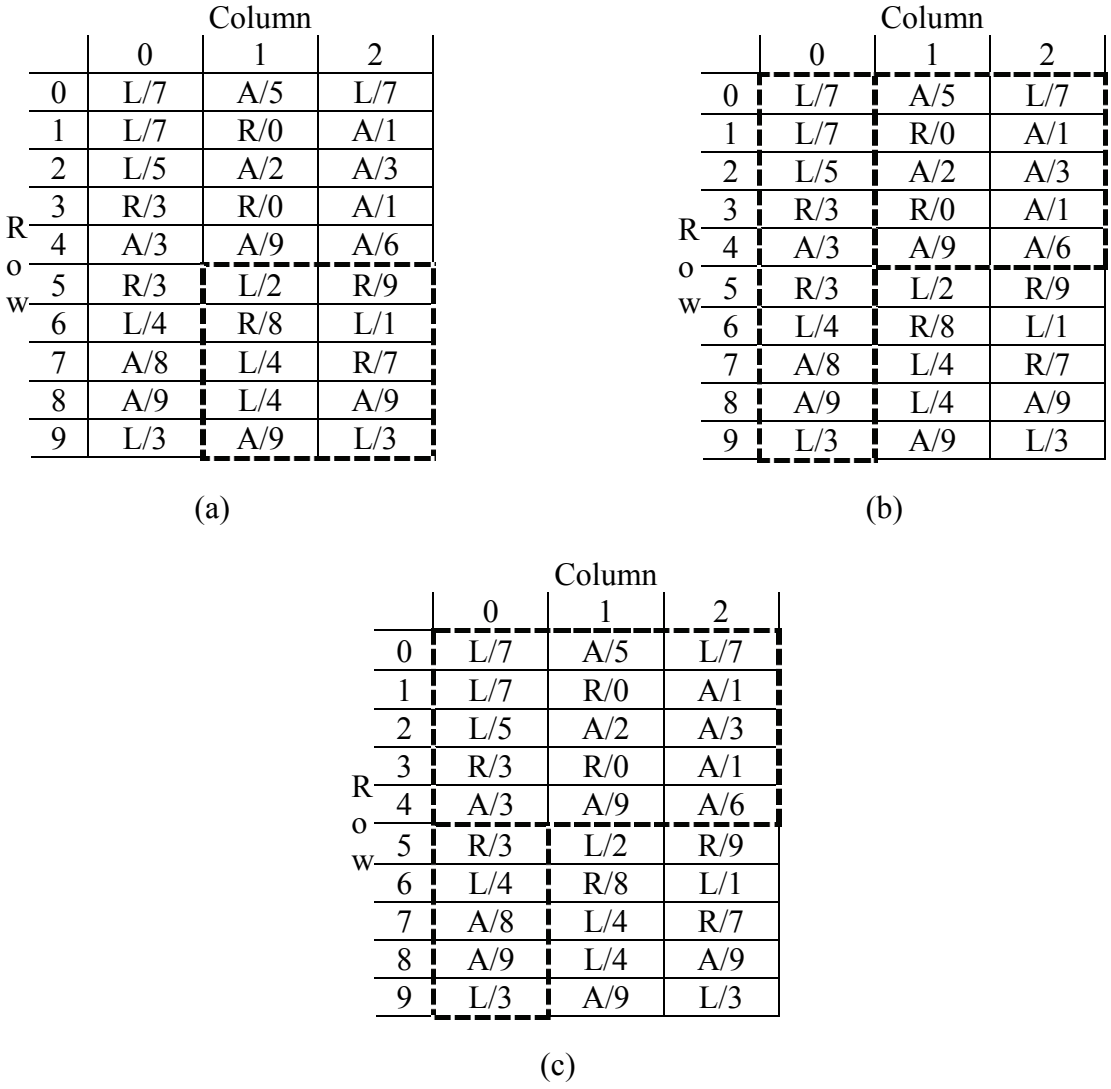


Figure 4.53 - Determining the 2 sub-areas.

The parameter *choice* contains either the constant value COLS_DIFF or the constant value ROWS_DIFF to specify the pair of sub-areas chosen with the value passed back out to the calling method via call by reference. The parameters area1Rows and

area1Cols contain the number of rows and number of columns, respectively, of the area 1 chosen with the values passed back out to the calling method via call by reference. The parameters area2Rows and area2Cols contain the number of rows and number of columns, respectively, of the area 2 chosen with the value passed back out to the calling method via call by reference.

The method determineSubAreas chooses the way to divide the remaining area of mainArea by comparing the size of COLS_DIFF's area 1 and the size of ROWS_DIFF's area 2. If the size of COLS_DIFF's area 1 is less than the size of ROWS_DIFF's area 2 (line 8, figure 4.52) then choose COLS_DIFF by executing the then clause at lines 9 to 15 in figure 4.52. Line 10 in figure 4.52 assigns *choice* the value COLS_DIFF. Lines 11 and 12 in figure 4.52 assign area1Rows and area1Cols the dimensions of COLS_DIFF's area 1. Lines 13 and 14 in figure 4.52 assign area2Rows and area2Cols the dimensions of COLS_DIFF's area 2. Otherwise, the choose ROWS_DIFF by executing the else clause at lines 16 to 23 in figure 4.52. Line 18 in figure 4.52 assigns choice the value ROWS_DIFF. Lines 19 and 20 in figure 4.52 assign area1Rows and area1Cols the dimensions of ROWS_DIFF's area 1. Lines 21 and 22 in figure 4.52 assign area2Rows and area2Cols the dimensions of ROWS_DIFF's area 2. For the example in figure 4.53, the method determineSubAreas chooses the COLS_DIFF way of dividing the remaining area of mainArea into two smaller sub-areas as shown in figure 4.53b. The next subsection describes the next auxiliary method used by the main method crossoverPartitions.

4.6.3 The Method computeArea1Coord

The method computeArea1Coord, shown in figure 4.54, computes the coordinates of the top left corner of the area 1 chosen by the method determineSubAreas. The parameter *corner* specifies which corner anchors the new crossover point within the rectangular portion of the agent, referred to here by the name mainArea. The parameter *choice* identifies the way, COLS_DIFF or ROWS_DIFF, the method determineSubAreas divided the remaining area of mainArea into two sub-areas, area 1 and area 2.

```
 1: void FSM::computeArea1Coord(int corner, int choice,
 2:                             int xPos, int yPos,
 3:                             int newRowsDiff,
 4:                             int newCols,
 5:                             int &area1XPos,
 6:                             int &area1YPos)
 7: {
 8:   area1XPos = xPos;
 9:   area1YPos = yPos;
10:   switch (corner) {
11:     case TOP_LFT: area1XPos += newCols;
12:                   break;
13:     case BOT_LFT: area1XPos += newCols;
14:     case BOT_RGT: if (choice == ROWS_DIFF)
15:                   {
16:                     area1YPos += newRowsDiff;
17:                   }
18:                   break;
19:   }
20: }
```

Figure 4.54 - The method to compute sub-area 1's coordinates.

The parameters xPos and yPos are the coordinates of the top left corner of mainArea. The parameter newRowsDiff is the difference between the number of rows in mainArea and the number of rows in the new crossover point. The parameter newCols is

the number of columns of the new crossover point created inside mainArea. The parameters area1XPos and area1YPos are the coordinates of the top left corner of area 1 passed back out to the calling method via call by reference.

Recall figure 4.53a illustrates an example of a mainArea with 10 rows and 3 columns, the new crossover point with 5 rows and 2 columns anchored at the bottom right corner of mainArea, and the method determineSubAreas choosing the COLS_DIFF way of dividing the remaining area of mainArea into two smaller sub-areas as shown in figure 4.53b. For the example in figure 4.53b, the parameter *corner* is BOT_RGT, *choice* is COLS_DIFF, xPos is 0, yPos is 0, newRowsDiff is 5 and newCols is 2.

Since area 1 is inside mainArea, the coordinates of area 1's top left corner are equal to the coordinates of mainArea's top left corner with a non-negative offset added to each coordinate. Line 8 in figure 4.54 assigns the xPos coordinate of mainArea to the area1XPos coordinate of area 1. Line 9 in figure 4.54 assigns the yPos coordinate of mainArea to the area1YPos coordinate of area 1. For the example in figure 4.53b, mainArea's top left coordinates are (0, 0) and area 1's top left coordinates are (0, 0) after executing lines 8 and 9 in figure 4.54. The "switch" statement at lines 10 to 19 in figure 4.54 adds the appropriate offset to the appropriate area 1 coordinate. For the example in figure 4.53b, the "switch" statement executes the case at lines 14 through 18 in figure 4.54 because the parameter *corner* is BOT_RGT. The case at lines 14 through 17 in figure 4.54 adds the non-negative offset newRowsDiff to the y coordinate for area 1, area1YPos, only if the parameter *choice* contains the value ROWS_DIFF. Since the parameter *choice* is COLS_DIFF for the example in figure 4.53b, area1YPos remains unchanged resulting in the coordinates (area1XPos, area1YPos) of the top left corner of

area 1 as (0, 0). The next subsection describes the next auxiliary method used by the main method crossoverPartitions.

4.6.4 The Method computeArea2Coord

The method computeArea2Coord, shown in figure 4.55, computes the coordinates of the top left corner of the area 2 chosen by the method determineSubAreas. The parameter *corner* specifies which corner anchors the new crossover point within the rectangular portion of the agent, referred to here by the name mainArea. The parameter *choice* identifies the way, COLS_DIFF or ROWS_DIFF, the method determineSubAreas divided the remaining area of mainArea into two sub-areas, area 1 and area 2.

```
 1: void FSM::computeArea2Coord(int corner, int choice,
 2:                             int xPos, int yPos,
 3:                             int newRows,
 4:                             int newColsDiff,
 5:                             int &area2XPos,
 6:                             int &area2YPos)
 7: {
 8:   area2XPos = xPos;
 9:   area2YPos = yPos;
10:   switch (corner) {
11:     case TOP_LFT: area2YPos += newRows;
12:                   break;
13:     case TOP_RGT: area2YPos += newRows;
14:     case BOT_RGT: if (choice == COLS_DIFF)
15:                   {
16:                     area2XPos += newColsDiff;
17:                   }
18:                   break;
19:   }
20: }
```

Figure 4.55 - The method to compute sub-area 2's coordinates.

The parameters xPos and yPos are the coordinates of the top left corner of mainArea. The parameter newRows is the number of rows of the new crossover point created inside mainArea. The parameter newColsDiff is the difference between the number of columns in mainArea and the number of columns in the new crossover point. The parameters area2XPos and area2YPos are the coordinates of the top left corner of area 2 passed back out to the calling method via call by reference.

Recall figure 4.53a illustrates an example of a mainArea with 10 rows and 3 columns, the new crossover point with 5 rows and 2 columns anchored at the bottom right corner of mainArea, and the method determineSubAreas choosing the COLS_DIFF way of dividing the remaining area of mainArea into two smaller sub-areas as shown in figure 4.53b. For the example in figure 4.53b, the parameter *corner* is BOT_RGT, *choice* is COLS_DIFF, xPos is 0, yPos is 0, newRows is 5 and newColsDiff is 1.

Since area 2 is inside mainArea, the coordinates of area 2's top left corner are equal to the coordinates of mainArea's top left corner with a non-negative offset added to each coordinate. Line 8 in figure 4.55 assigns the xPos coordinate of mainArea to the area2XPos coordinate of area 2. Line 9 in figure 4.55 assigns the yPos coordinate of mainArea to the area2YPos coordinate of area 2. For the example in figure 4.53b, mainArea's top left coordinates are (0, 0) and area 2's top left coordinates are (0, 0) after executing lines 8 and 9 in figure 4.55. The "switch" statement at lines 10 to 19 in figure 4.55 adds the appropriate offset to the appropriate area 2 coordinate. For the example in figure 4.53b, the "switch" statement executes the case at lines 14 through 18 in figure 4.55 because the parameter *corner* is BOT_RGT. The case at lines 14 through 17 in figure 4.55 adds the non-negative offset newColsDiff to the x coordinate for area 2,

area2XPos, only if the parameter *choice* contains the value COLS_DIFF. Since the parameter *choice* is COLS_DIFF for the example in figure 4.53b, add the offset newColsDiff to area2XPos to compute the coordinates (area2XPos, area2YPos) of the top left corner of area 2 as (1, 0). The next subsection describes the next auxiliary method used by the main method crossoverPartitions.

4.6.5 The Method insertSubArea

The method insertSubArea, shown in figure 4.56, inserts a sub-area into the linked list of sorted partitions. The parameters areaRows and areaCols are the number of rows and number of columns, respectively, of the sub-area. The parameters areaXPos and areaYPos are the coordinates of the top left corner of the sub-area. The parameter sortedParts is the linked list of sorted partitions.

```
 1: void FSM::insertSubArea
 2:              (int areaRows, int areaCols,
 3:               int areaXPos, int areaYPos,
 4:               LinkedList<CrossoverArea> &sortedParts)
 5: {
 6:   CrossoverArea subArea;
 7:
 8:   if ((areaRows*areaCols) > 0)
 9:   {
10:     subArea.setCrossoverArea(areaRows, areaCols,
11:                              areaXPos, areaYPos);
12:     sortedParts.insertSorted(subArea);
13:   }
14: }
```

Figure 4.56 - The method to insert a sub-area into the sorted partitions linked list.

In subsection 4.6.1, the assumption is the remaining area of mainArea can partition into 2 sub-areas in order to simplify the process for dealing with the remaining

area of mainArea. If the sub-area's area is greater than 0 (line 8, figure 4.56) then the sub-area actually exists for insertion into the linked list of sorted partitions, sortedParts. The variable subArea (line 6, figure 4.56) is an instance of the CrossoverArea class to store the sub-area for insertion into the linked list sortedParts. The call to the setCrossoverArea method (lines 10 and 11, figure 4.56) of the CrossoverArea class stores inside the CrossoverArea variable subArea the values defining the sub-area: areaRows, areaCols, areaXPos and areaYPos. A call to the insertSorted method (line 12, figure 4.56) of the LinkedList class, passing the variable subArea as a parameter, inserts a copy of the variable subArea into the LinkedList variable sortedParts. The next subsection describes the last auxiliary method used by the main method crossoverPartitions.

4.6.6 The Method computeNewCoord

The method computeNewCoord, shown in figure 4.57, computes the coordinates of the top left corner of the new crossover point. The parameter *corner* specifies which corner anchors the new crossover point within the rectangular portion of the finite state automation, referred to here by the name mainArea. The parameters xPos and yPos are the coordinates of the top left corner of mainArea. The parameter newRowsDiff is the difference between the number of rows in mainArea and the number of rows in the new crossover point. The parameter newColsDiff is the difference between the number of columns in mainArea and the number of columns in the new crossover point. The parameters newXPos and newYPos are the coordinates of the top left corner of the new crossover point passed back out to the calling method via call by reference.

```
 1: void FSM::computeNewCoord(int corner, int xPos,
 2:                           int yPos, int newRowsDiff,
 3:                           int newColsDiff,
 4:                           int &newXPos, int &newYPos)
 5: {
 6:   switch (corner) {
 7:     case TOP_LFT: newXPos = xPos;
 8:                   newYPos = yPos;
 9:                   break;
10:     case TOP_RGT: newXPos = xPos + newColsDiff;
11:                   newYPos = yPos;
12:                   break;
13:     case BOT_LFT: newXPos = xPos;
14:                   newYPos = yPos + newRowsDiff;
15:                   break;
16:     case BOT_RGT: newXPos = xPos + newColsDiff;
17:                   newYPos = yPos + newRowsDiff;
18:                   break;
19:   }
20: }
```

Figure 4.57 - The method to compute the new crossover point's coordinates.

Recall figure 4.53a illustrates an example of a mainArea with 10 rows and 3 columns, and the new crossover point with 5 rows and 2 columns anchored at the bottom right corner of mainArea. The parameter *corner* is BOT_RGT, xPos is 0, yPos is 0, newRowsDiff is 5, and newColsDiff is 1. Since the new crossover point is inside mainArea, the coordinates of the new crossover point's top left corner are equal to the coordinates of mainArea's top left corner with a non-negative offset added to each coordinate. The "switch" statement at lines 6 to 19 in figure 4.57 computes the coordinates of the new crossover point's top left corner based upon the value of the parameter *corner*. For the example in figure 4.53a, the "switch" statement executes the case at lines 16 through 18 in figure 4.57 because the parameter *corner* is BOT_RGT. Line 16 in figure 4.57 adds the non-negative offset newColsDiff to xPos to compute the x coordinate for the new crossover point, newXPos. Line 17 in figure 4.57 adds the non-

negative offset newRowsDiff to yPos to compute the y coordinate for the new crossover point, newYPos. Therefore, the coordinates (newXPos, newYPos) of the new crossover point in figure 4.53a are (1, 5). The next section provides a detailed description of the methods to execute and tune the signature recognition genetic algorithm.

4.7 The Signature Recognition Genetic Algorithm

This section provides a detailed description of the methods for the signature recognition genetic algorithm. The next subsection describes the main method geneticAlgorithm to execute the signature recognition genetic algorithm. Subsection 4.7.2 describes the main method to tune the crossover and mutation rates for this genetic algorithm.

4.7.1 The Genetic Algorithm Description

The genetic algorithm for this experiment evolves a population of agents, searching for an agent or agents that can traverse the entire length of the current path in the grid. The method geneticAlgorithm in figure 4.58 executes the signature recognition program's genetic algorithm for the current path in the grid. The parameter pathNum provides the number of the current path in the grid. For each path, the genetic algorithm executes a fixed number of times. The parameter runNum is the cardinal number representing the number of times the genetic algorithm executed for the current path in the grid once the current call to the method geneticAlgorithm completes execution. The parameter *world* is a pointer to a Grid object which contains the grid. The parameter theAgents is a pointer to a Population object containing the entire population of agents.

The parameter outFile represents the output file used by the genetic algorithm. The last parameter maxNumGen represents the maximum number of generations for execution of the genetic algorithm.

```
 1: void geneticAlgorithm(int pathNum, int runNum,
 2:                       Grid *world,
 3:                       Population *theAgents,
 4:                       ofstream outFile, int maxNumGen)
 5: {
 6:   bool terminate;
 7:   int i;
 8:
 9:   terminate =
10:     theAgents->computeEachFitness(world, outFile,
11:                                   pathNum, runNum, 0);
12:   for (i=1; i<=maxNumGen && !terminate; i++)
13:   {
14:     theAgents->createNewPop();
15:     terminate =
16:       theAgents->computeEachFitness(world, outFile,
17:                                     pathNum, runNum,
18:                                     i);
19:   }
20: }
```

Figure 4.58 - The code for the genetic algorithm method.

The genetic algorithm begins at lines 9 through 11 in figure 4.58 by computing the fitness value for each member of the population via a call to the computeEachFitness method of the Population class passing *world*, outFile, pathNum, runNum and the generation number (0) as parameters. The method computeEachFitness returns a value of true if at least one agent in the current population has perfect fitness; otherwise computeEachFitness returns a value of false. An agent has perfect fitness when its fitness value is equal to the length of the path in the grid hence the agent consumed the entire path. The genetic algorithm assigns the value returned by computeEachFitness to the

Boolean variable *terminate*. In addition, the method computeEachFitness writes to the output file, via the outFile parameter, each agent with perfect fitness.

The "for" loop at lines 12 through 19 in figure 4.58 generates generation after generation of agents until the genetic algorithm reaches the maximum number of generations or at least one agent in the current population has perfect fitness. The variable *i* stores the current generation number and initializes to generation 1 at line 12 in figure 4.58. A call (line 14, figure 4.58) to the createNewPop method of the Population class creates the next generation of agents. The next generation of agents becomes the population for the current generation. The genetic algorithm computes the fitness value of each member of the population via a call (lines 15 to 18, figure 4.58) to the computeEachFitness method of the Population class passing *world*, outFile, pathNum, runNum and the generation number (*i*) as parameters. The method computeEachFitness returns a value of true if at least one agent in the current population has perfect fitness; otherwise computeEachFitness returns a value of false. The genetic algorithm assigns the value returned by computeEachFitness to the Boolean variable *terminate*. In addition, the method computeEachFitness writes to the output file, via the outFile parameter, each agent with perfect fitness along with the path number, run number and generation number producing the perfectly fit agent. The "for" loop (the genetic algorithm) terminates (line 12, figure 4.58) when *i* is greater than maxNumGen (the maximum number of generations) or the Boolean variable *terminate* is true. The next subsection describes the main method to tune the crossover and mutation rates for this genetic algorithm.

4.7.2 Tuning the Genetic Algorithm

The signature recognition genetic algorithm tuning program chooses the optimal crossover and mutation rate pair for the signature recognition program from one of the pairings of a crossover rate and a mutation rate: [5%, 6%, 7%, 8%, 9%, 10%]×[1%, 2%, 3%]. The tuning program uses five paths of various lengths (562, 1224, 763, 1126 and 503) executing the signature recognition genetic algorithm once for each path and each crossover and mutation rate pair. Recall from subsection 4.7.1 that the signature recognition genetic algorithm writes to the output file, each agent with perfect fitness along with the path number, run number and generation number producing the perfectly fit agent. An examination of the output file produced by the signature recognition genetic algorithm tuning program reveals the crossover and mutation pair that generates agents with perfect fitness in the least number of generations. It is the pair of rates in the least number of generations that the signature recognition program uses in the main experiments.

```
 1: #include <iostream.h>
 2: #include <fstream.h>
 3: #include "..\Common\FSM.h"
 4: #include "..\Common\Grid.h"
 5: #include "..\Common\Population.h"
 6:
 7: int main()
 8: {
 9:   bool terminate;
10:   int i, pathNum;
11:   int gridRows, gridColumns;
12:   int numInputs, numOutputs, numStates;
13:   int popSize, maxNumGen, maxMoveMult;
14:   float crossoverRate, mutationRate;
15:   ifstream inFile;
```

```
16:   ofstream outFile;
17:   Grid *world;
18:   Population *theAgents;
19:
20:   inFile.open("parameters.txt", ios::in);
21:   inFile >> gridRows >> gridColumns;
22:   inFile >> numInputs >> numOutputs >> numStates;
23:   inFile >> popSize >> maxNumGen >> maxMoveMult;
24:   inFile.close();
25:
26:   inFile.open("paths001.txt", ios::in);
27:   outFile.open("TuneGA.txt", ios::out);
28:   FSM::setFSM(numInputs, numOutputs, numStates, 0, 0);
29:   world = new Grid(gridRows, gridColumns);
30:   theAgents = new Population(popSize, true);
31:
32:   for (pathNum=1; pathNum<=5; pathNum++)
33:   {
34:     world->addNewPathToGrid(maxMoveMult, inFile);
35:     for (crossoverRate=5; crossoverRate<=10;
36:          crossoverRate++)
37:     {
38:       for (mutationRate=1; mutationRate<=3;
39:            mutationRate++)
40:       {
41:         outFile << "Crossover rate: ";
42:         outFile << crossoverRate << "%, ";
43:         outFile << "Mutation rate: ";
44:         outFile << mutationRate << "%";
45:         outFile << endl;
46:         theAgents->initializePopulation(crossoverRate,
47:                                         mutationRate);
48:         geneticAlgorithm(pathNum, 1, world, theAgents,
49:                          outFile, maxNumGen);
50:       }
51:     }
52:   }
53:   inFile.close();
54:   outFile.close();
55:
56:   return 0;
57: }
```

Figure 4.59 - The program to tune the genetic algorithm's crossover and mutation rates.

Figure 4.59 lists the code for the signature recognition genetic algorithm tuning program. The statements at lines 1 and 2 in figure 4.59 include, into the signature recognition genetic algorithm tuning program, the header files for the standard C++ input and output libraries. The statements at lines 3 through 5 in figure 4.59 include, into the signature recognition genetic algorithm tuning program, the header files for the user defined classes FSM, Grid and Population. The FSM class implements the agent, the Grid class implements the grid containing the path, and the Population class implements the entire population of agents for the signature recognition genetic algorithm. The function main (lines 7 to 57, figure 4.59) is the starting point for the execution of the signature recognition genetic algorithm tuning program. Line 20 in figure 4.59 opens the parameters file, parameters.txt. Lines 21 through 23 in figure 4.59 read the following signature recognition genetic algorithm parameters (section 4.1) gridRows, gridColumns, numInputs, numOutputs, numStates, popSize, maxNumGen and maxMoveMult from the parameters file, parameters.txt. The parameters file closes at line 24 in figure 4.59.

The paths file (path001.txt) opens for input at line 26 in figure 4.59 and the file TuneGA.txt opens for output at line 27 in figure 4.59 to store the agents with perfect fitness produced by runs of the signature recognition genetic algorithm tuning program. A call (line 28, figure 4.59) to the setFSM method of the FSM class passing numInputs, numOutputs, numStates, the crossover rate 0 and mutation rate 0, as parameters initializes each agent in the population with these five values. At line 29 in figure 4.59, the variable *world* is a Grid object whose constructor initializes the grid with gridRows number of rows and gridColumns number of columns. At line 30 in figure 4.59, the variable theAgents is a Population object whose constructor creates the population of

agents. The first parameter of the Population constructor specifies the size of the population, popSize. The second parameter of the Population constructor is a Boolean variable to indicate to the constructor, with a value of true, to maintain a copy of the starting population of agents so that the execution of the signature recognition genetic algorithm, for each crossover and mutation rate pair, begins with the same initial population.

The "for" loop at lines 32 through 52 in figure 4.59 controls the execution of the signature recognition genetic algorithm tuning program on each of the first five paths in the input file paths001.txt. A call to the method addNewPathToGrid of the Grid class at line 34 in figure 4.59 reads the next path in the file paths001.txt and re-creates the path in the grid. The maximum number of moves multiplier, maxMovesMult, and the file variable inFile, for the input file paths001.txt, pass their values into the method addNewPathToGrid via the method's parameters. The "for" loop at lines 35 through 51 in figure 4.59 controls the iteration of the variable crossoverRate beginning with the value of 5 and increasing crossoverRate's value by 1 until crossoverRate is greater than 10. Nested inside the "for" loop controlling crossoverRate is the "for" loop at lines 38 through 50 in figure 4.59 which controls the iteration of the variable mutationRate beginning with the value of 1 and increasing mutationRate's value by 1 until mutationRate is greater than 3. Nested inside the "for" loop controlling mutationRate is the main execution of the signature recognition genetic algorithm tuning program. Lines 41 through 45 in figure 4.59 print the crossover and mutation rates to one line in the output file TuneGA.txt. The method initializePopulation of the Population class (lines 46 and 47, figure 4.59) copies the stored starting population of agents to the initial

population of agents for the signature recognition genetic algorithm executed in the next statement of this method. The method initializePopulation receives the crossover and mutation rates as parameters in order to assign the crossover and mutation rates to each agent in the population. The genetic algorithm executes by a call to the geneticAlgorithm method at lines 48 and 49 in figure 4.59 passing pathNum, 1, *world*, theAgents, outFile and maxNumGen as parameters. After the genetic algorithm executes for each path and crossover and mutation rate pair, the tuning program reaches lines 53 and 54 in figure 4.59 to close the input file paths001.txt and the output file TuneGA.txt. Finally, the main method terminates by executing a return statement (line 56, figure 4.59) to return a value of 0. The next section describes the main methods for the signature recognition program.

4.8 The Signature Recognition Program Description

The program description begins in the next subsection 4.8.1 with the program's main method. Subsection 4.8.2 describes the auxiliary methods to create the names of the three files used by the program. Subsection 4.8.3 describes the auxiliary method computePathAndRunNums. Subsection 4.8.4 describes the auxiliary method orientPathsFile.

4.8.1 The Signature Recognition Program Main Method

Figure 4.60 lists the code for the main method of the signature recognition program. The statement at line 1 in figure 4.60 includes, into the signature recognition program, the header file for the standard C++ windows library. This line is in the signature recognition program only if the compiled program executes on a computer

running the Microsoft Windows operating system. The statements at lines 2 and 3 in figure 4.60 include, into the signature recognition program, the header files for the standard C++ input and output libraries. The statements at lines 4 through 6 in figure 4.60 include, into the signature recognition program, the header files for the user defined classes FSM, Grid and Population. The FSM class implements the agent, the Grid class implements the grid containing the path, and the Population class implements the entire population of agents for the signature recognition program.

```
 1: #include <windows.h>
 2: #include <iostream.h>
 3: #include <fstream.h>
 4: #include "..\Common\FSM.h"
 5: #include "..\Common\Grid.h"
 6: #include "..\Common\Population.h"
 7:
 8: int main()
 9: {
10:   bool terminate;
11:   char ch, pathsFilename[13], outputFilename[10],
12:   char statusFilename[14];
13:   int i, j;
14:   int gridRows, gridColumns;
15:   int numInputs, numOutputs, numStates;
16:   int popSize, maxNumGen;
17:   int numRuns, maxMoveMult, fileNum;
18:   int pathNum, initialRunNum;
19:   float crossoverRate, mutationRate;
20:   ifstream parametersFile, pathsFile;
21:   ofstream outFile, statusFile;
22:   Grid *world;
23:   Population *theAgents;
24:
25:   SetPriorityClass(GetCurrentProcess(),
26:                    IDLE_PRIORITY_CLASS);
27:
28:   parametersFile.open("parameters.txt", ios::in);
29:   parametersFile >> gridRows >> gridColumns;
30:   parametersFile >> numInputs >> numOutputs;
31:   parametersFile >> numStates >> popSize >> maxNumGen;
32:   parametersFile >> numRuns >> maxMoveMult;
```

```
33:   parametersFile >> crossoverRate >> mutationRate;
34:   parametersFile >> fileNum;
35:   parametersFile.close();
36:
37:   createPathsFilename(pathsFilename, fileNum);
38:   createOutputFilename(outputFilename, fileNum);
39:   createStatusFilename(statusFilename, fileNum);
40:   pathsFile.open(pathsFilename, ios::in);
41:   outFile.open(outputFilename, ios::app);
42:   statusFile.open(statusFilename, ios::app);
43:   FSM::setFSM(numInputs, numOutputs, numStates,
44:               crossoverRate, mutationRate);
45:   world = new Grid(gridRows, gridColumns);
46:   theAgents = new Population(popSize, false);
47:
48:   computePathAndRunNums(statusFilename, numRuns,
49:                         pathNum, initialRunNum);
50:   if (pathNum > 1)
51:     orientPathsFile(pathsFile, pathNum-1);
52:   while (pathsFile.get(ch))
53:   {
54:     world->addNewPathToGrid(maxMoveMult, pathsFile);
55:     for (i=initialRunNum; i<=numRuns; i++)
56:     {
57:       theAgents->initializePopulation();
58:       geneticAlgorithm(pathNum, i, world, theAgents,
59:                        outFile, maxNumGen);
60:       statusFile << pathNum << " " << i << endl;
61:     }
62:
63:     pathNum++;
64:     initialRunNum = 1;
65:   }
66:   pathsFile.close();
67:   outFile.close();
68:   statusFile.close();
69:
70:   return 0;
71: }
```

Figure 4.60 - The signature recognition program's main method.

The main method (lines 8 to 71, figure 4.60) is the starting point for the execution of the signature recognition program. The statement at lines 25 and 26 in figure 4.60 sets

the signature recognition program's execution priority to adjust program execution during the entire day. When no one uses the computer, the signature recognition program's priority sets to high. When someone uses the computer, the signature recognition program's priority sets to low. This line is in the signature recognition program only if the compiled program executes on a computer running the Microsoft Windows operating system. The parameters file (parameters.txt) opens for input at line 28 in figure 4.60. Lines 29 through 34 in figure 4.60 read the following signature recognition parameters (section 4.1): gridRows, gridColumns, numInputs, numOutputs, numStates, popSize, maxNumGen, numRuns, maxMoveMult, crossoverRate, mutationRate and fileNum from the parameters file parameters.txt. The parameters file closes at line 35 in figure 4.60.

A call (line 37, figure 4.60) to the createPathsFilename method, passing pathsFilename and fileNum as parameters, creates the filename for the file containing the paths. A call (line 38, figure 4.60) to the createOutputFilename method, passing outputFilename and fileNum as parameters, creates the filename for the file containing the signature recognition program results. A call (line 39, figure 4.60) to the createStatusFilename method, passing statusFilename and fileNum as parameters, creates the filename for the file containing the completed path number and run number pairs.

The paths file opens for input at line 40 in figure 4.60 and the output file opens for output at line 41 in figure 4.60 to store the agents with perfect fitness produced by various runs of the signature recognition program. The status file opens for input/output at line 42 in figure 4.60 to obtain as well as record the paths and runs completed by the signature recognition program. A call (lines 43 and 44, figure 4.60) to the setFSM method of the FSM class, passing numInputs, numOutputs, numStates, crossoverRate and

mutationRate as parameters, initializes each agent in the population with these five values. At line 45 in figure 4.60, the variable *world* is a Grid object whose constructor initializes the grid with gridRows number of rows and gridColumns number of columns. At line 46 in figure 4.60, the variable theAgents is a Population object whose constructor creates the population of agents. The first parameter of the Population constructor specifies the size of the population, popSize. The second parameter of the Population constructor is a Boolean variable to indicate to the constructor, with a value of false, that each execution of the signature recognition genetic algorithm begins with a different randomly chosen initial population.

The call to the computePathAndRunNums method at lines 48 and 49 in figure 4.60, passing statusFilename, numRuns, pathNum and initialRunNum as parameters, determines, from the status file, the initial run of the current path to begin the signature recognition program execution. The current path number passes back out via the pass by reference pathNum parameter and the initial run number passes back out via the pass by reference initialRunNum parameter. If the current path number (pathNum) is greater than 1 then a call (line 51, figure 4.60) to the orientPathsFile, passing pathsFile and pathNum-1 as parameters, has the signature recognition program read past the first pathNum-1 paths in the paths file such that the program begins execution with the pathNum path. Each path line in the paths file begins with a space character allowing the signature recognition program an easy way to detect the end of the paths file. At line 52 in figure 4.60, the call to the *get* method attempts to read a character from the paths file placing the character in the variable *ch*. The *get* method returns true if it read a character from the file and false otherwise. A true value means that the *get* method read a space

character and the program can read another path from the paths file. A false value means the program reached the end of the paths file and can terminate execution.

The "while" loop (lines 52 to 65, figure 4.60) contains the main processing of the signature recognition program. At line 54 in figure 4.60, a call to the method addNewPathToGrid of the Grid class reads the next path in the paths file and re-creates the path in the grid. The maximum number of moves multiplier, maxMovesMult, and the file variable pathsFile pass their values into the method addNewPathToGrid via the method's parameters. The "for" loop at lines 55 through 61 in figure 4.60 controls the iteration of the variable *i* beginning with the value of initialRunNum and increasing *i*'s value by 1 until *i* is greater than numRuns. The method initializePopulation of the Population class (line 57, figure 4.60) randomly creates an initial population of agents for the signature recognition genetic algorithm executed in the next statement. The signature recognition genetic algorithm executes (lines 58 and 59, figure 4.60) by a call to the geneticAlgorithm method (subsection 4.7.1) passing pathNum, *i*, *world*, theAgents, outFile and maxNumGen as parameters. After the signature recognition genetic algorithm terminates, the signature recognition program writes (line 60, figure 4.60) the path number (pathNum) and run number (*i*) just completed to the status file.

All the runs (numRuns) of the current path complete when the "for" loop terminates execution. Line 63 of figure 4.60 increases the path number (pathNum) by 1. The execution of the signature recognition program may terminate unexpectedly in the middle of processing a particular run of a path. When the signature recognition program resumes execution, the program continues execution with the next run number of the current path found by the computePathAndRunNums method. This next run number

initializes the initialRunNum variable which maybe a value between 1 and numRuns. But, for all remaining paths processed by the program, the initial run number must be 1. Therefore, line 64 of figure 4.60 sets the initial run number (initialRunNum) to 1. After the "while" loop terminates, lines 66 through 68 in figure 4.60 closes the paths, output and status files by calls to the file close method. Finally, the main method terminates by executing a return statement (line 70, figure 4.60) to return a value of 0. The next subsection describes the methods to create the three filenames used by the signature recognition program.

4.8.2 The Filename Creation Methods

A simple form of parallelism expedites the execution of the signature recognition experiments. The path creation program, using the path creation methods of section 4.4, creates a data set of 10,000 paths in groups of 100 with each group placed in a separate file. A major portion of each group's filename is the same set of characters. Each filename begins with the string "paths". The next three characters in the filename are digits corresponding to the group number, a number between 1 and 100. Single and double digit group numbers use leading zeros. The filename ends with the extension ".txt". As an example, group 1's paths store in the file with the name paths001.txt.

For each signature recognition experiment, 100 copies of the signature recognition program execute in parallel with each copy processing a unique group of paths. The method createPathsFilename in figure 4.61 creates the paths filename in the 13 character array parameter pathsFilename where the fileNum parameter specifies the unique group number for the particular copy of the signature recognition program.

```
void createPathsFilename(char *pathsFilename, int fileNum)
{
  pathsFilename[0] = 'p';
  pathsFilename[1] = 'a';
  pathsFilename[2] = 't';
  pathsFilename[3] = 'h';
  pathsFilename[4] = 's';
  pathsFilename[5] = fileNum / 100 % 10 + '0';
  pathsFilename[6] = fileNum / 10 % 10 + '0';
  pathsFilename[7] = fileNum % 10 + '0';
  pathsFilename[8] = '.';
  pathsFilename[9] = 't';
  pathsFilename[10] = 'x';
  pathsFilename[11] = 't';
  pathsFilename[12] = '\0';
}
```

Figure 4.61 - The method to create the filename for the file containing the paths.

```
void createOutputFilename(char *outputFilename,
                          int fileNum)
{
  outputFilename[0] = 'g';
  outputFilename[1] = 'a';
  outputFilename[2] = fileNum / 100 % 10 + '0';
  outputFilename[3] = fileNum / 10 % 10 + '0';
  outputFilename[4] = fileNum % 10 + '0';
  outputFilename[5] = '.';
  outputFilename[6] = 't';
  outputFilename[7] = 'x';
  outputFilename[8] = 't';
  outputFilename[9] = '\0';
}
```

Figure 4.62 - The method to create the filename for the file containing the program results.

The same procedure creates the name for the output file containing the results for the particular copy of the signature recognition program. The method createOutputFilename in figure 4.62 creates the output filename in the 10 character array parameter outputFilename where the fileNum parameter specifies the unique group

number for the particular copy of the signature recognition program. Each filename begins with the string “ga”. The next three characters in the filename are the digits corresponding to the group number. The filename ends with the extension “.txt”. As an example, group 1’s output filename is ga001.txt.

```
void createStatusFilename(char *statusFilename,
                          int fileNum)
{
  statusFilename[0] = 's;
  statusFilename[1] = 't;
  statusFilename[2] = 'a';
  statusFilename[3] = 't;
  statusFilename[4] = 'u;
  statusFilename[5] = 's;
  statusFilename[6] = fileNum / 100 % 10 + '0';
  statusFilename[7] = fileNum / 10 % 10 + '0';
  statusFilename[8] = fileNum % 10 + '0';
  statusFilename[9] = '.';
  statusFilename[10] = 't';
  statusFilename[11] = 'x';
  statusFilename[12] = 't';
  statusFilename[13] = '\0';
}
```

Figure 4.63 - The method to create the filename for the file containing the completed path number and run number pairs.

The same procedure creates the name for the file containing the completed path number and run number pairs for the particular copy of the signature recognition program. The method createStatusFilename in figure 4.63 creates the status filename in the 14 character array parameter statusFilename where the fileNum parameter specifies the unique group number for the particular copy of the signature recognition program. Each filename begins with the string “status”. The next three characters in the filename are the digits corresponding to the group number. The filename ends with the extension “.txt”. As an example, group 1’s status filename is status001.txt. The next subsection

describes the next auxiliary method used by the signature recognition program's main method.

4.8.3 The Method computePathAndRunNums

The method computePathAndRunNums in figure 4.64 determines, from the status file, the initial run of the current path to begin execution of the signature recognition program. The first parameter, statusFilename, is the string representing the status file's filename. The parameter numRuns specifies the maximum number of genetic algorithm runs to perform on each path. The parameters pathNum and initialRunNum are the path number and initial run number, respectively, implemented via call by reference to pass the values back out to the calling method.

```
 1: void computePathAndRunNums(char *statusFilename,
 2:                             int numRuns,
 3:                             int &pathNum,
 4:                             int &initialRunNum)
 5: {
 6:   ifstream startFile;
 7:
 8:   startFile.open(statusFilename, ios::in);
 9:   while (!startFile.eof())
10:   {
11:     startFile >> pathNum >> initialRunNum;
12:   }
13:   startFile.close();
14:
15:   initialRunNum++;
16:   if (initialRunNum > numRuns)
17:   {
18:     pathNum++;
19:     initialRunNum = 1;
20:   }
21: }
```

Figure 4.64 - The method to find the initial run of the current path.

The statement at line 8 in figure 4.64 opens the status file for input. The "while" loop (lines 9 to 12, figure 4.64) executes until it reaches the end of the status file. Line 11 in figure 4.64 reads a pair of numbers into the parameters pathNum and initialRunNum. The last pair of numbers read into pathNum and initialRunNum is the last run of the current path completed by the signature recognition program during its last execution. The statement at line 13 in figure 4.64 closes the status file.

Line 15 of figure 4.64 increases the value of initialRunNum by 1 because the next run executed by the signature recognition program has a number one higher than the last run number read from the status file. If the initialRunNum value is greater than the value of numRuns (lines 16 to 20 in figure 4.64) then modify the values of pathNum and initialRunNum. In other words, the last run of the current path completed. Increase the value of pathRun by one (line 18, figure 4.64) and (line 19, figure 4.64) set the value of initialRunNum to 1.

4.8.4 The Method orientPathsFile

The method orientPathsFile in figure 4.65 reads past the first lineNum paths in the paths file positioning the signature recognition program to begin with path lineNum+1. The first parameter, pathsFilename, is the string representing the filename of the paths file. The parameter lineNum specifies the number of paths to read from the paths file. The "for" loop at lines 6 through 12 in figure 4.65 controls the iteration of the variable *i* beginning with the value of 1 and increasing *i*'s value by 1 until *i* is greater than lineNum. Each iteration of the "for" loop reads another line from the paths file by executing the "do-while" loop (lines 8 to 11, figure 4.65) inside the "for" loop. The "do-while" loop

reads each character of the line (line 9, figure 4.65) one at a time until the loop reaches the last character, the newline character. The next section details the batch file to install an instance of the signature recognition program on a computer running the Microsoft Windows operating system.

```
 1: void orientPathsFile(ifstream& pathsFile, int lineNum)
 2: {
 3:   char ch;
 4:   int i;
 5:
 6:   for (i=1; i<=lineNum; i++)
 7:   {
 8:     do {
 9:       pathsFile.get(ch);
10:     }
11:     while (ch != '\n');
12:   }
13: }
```

Figure 4.65 - The method to position the paths file to the next path to process.

4.9 The Program Execution under Microsoft Windows

For each of the five experiments, various instances of the signature recognition program ran on Windows computers. A Microsoft Visual C++ compiler provides an executable version of the signature recognition program and a batch file installs an instance of the signature recognition program for execution on a Windows computer. Figure 4.66 lists the batch file, ga.bat, to install an instance of the signature recognition program on a Windows computer. The Windows computer accesses the network drive containing the batch file and the "Code" and "Paths" directories. The Code directory contains the signature recognition program's executable file along with any other files

needed during the execution of the program. The Paths directory contains all 100 paths files of the data set used by the experiment, the non-crossover paths data set or the crossover paths data set.

```
1: mkdir "C:\Program Files\GA"
2: copy Code\*.* "C:\Program Files\GA\*.*"
3: copy Paths\paths%2.txt "C:\Program Files\GA\*.*"
4: echo %1 >> "C:\Program Files\GA\parameters.txt"
5: C:
6: cd "\Program Files\GA"
7: rename status.txt status%2.txt
8: instsrv GA "C:\Program Files\GA\srvany.exe"
9: GA.reg
```

Figure 4.66 - The sequence of commands in the batch file ga.bat.

As an example, the command “ga.bat 1 001” executes at the command prompt within a Windows command prompt window where the current directory is the root directory of the network drive. The batch file has two command line arguments represented by %1 and %2 within the batch file. The first command line argument (%1) is the group number for the paths processed by this instance of the signature recognition program. For the example above, the group number is 1. The second command line argument (%2) is the three digit version of the group number with leading zeros used for the single and double digit numbers. For the example above, the three digit version of the group number is 001.

The batch file command at line 1 in figure 4.66 creates a directory called “GA” inside the directory called “Program Files” on the computer’s hard drive, the computer’s ‘C’ drive. The batch file command on line 2 in figure 4.66 copies all of the files in the Code directory on the network drive to the GA directory on the C drive. The batch file command on line 3 in figure 4.66 copies the appropriate paths file from the Paths

directory on the network drive to the GA directory on the C drive. The copy command from line 3 uses the three digit number of the second command line argument (%2) to create the appropriate paths filename. For the example above, the appropriate paths filename is paths001.txt. The batch file command on line 4 in figure 4.66 appends the group number to the end of the contents of the file parameters.txt. The batch file command at line 5 in figure 4.66 switches from the network drive to the C drive. The batch file command at line 6 in figure 4.66 switches from the current directory on the C drive to the directory "\Program Files\GA". The batch file command at line 7 in figure 4.66 renames the file status.txt to the appropriate status filename. The rename command from line 7 uses the three digit number of the second command line argument (%2) to create the appropriate status filename. For the example above, the appropriate status filename is status001.txt.

In the Microsoft Windows operating system, a program running in the background is a service. A useful feature of a Windows service is that it runs in the background regardless of whether a user is or isn't logged into the computer. The two Microsoft Windows utility programs instsrv.exe and srvany.exe allow any program to execute as a service. The batch file command at line 8 in figure 4.66 installs the service called GA. The last batch file command at line 9 in figure 4.66 executes the Windows registry file GA.reg installing the registry keys needed by the signature recognition program to run as a service. The first registry key specifies the location and name of the signature recognition program's executable file. The second registry key specifies the directory containing all the signature recognition program files. The next section

provides a detailed description of the batch file and method to execute the signature recognition program on the computational cluster.

4.10 Programming in a Distributed Computing Environment

For each of the five experiments, various instances of the signature recognition program ran on the computational cluster. A Portland Group C++ compiler provides an executable version of the signature recognition program for execution on the cluster's computers. All 60 of the cluster's computers are accessible indirectly via a user account on the cluster's host computer running the Red Hat Linux operating system. The Sun Grid Engine (SGE) is the application that schedules a program for execution on one of the cluster's computers. The user's programs executing on the cluster computers use and share the disk storage of the user account's home directory. The execution of a program on the cluster takes the form of a shell script defined batch job. Each submission of a batch job to the SGE begins execution of another instance of the signature recognition program.

```
 1: #!/bin/bash
 2: #$ -S /bin/bash
 3: mkdir ga$2
 4: cd ga$2
 5: cp ../code/* .
 6: cp ../paths/paths$2.txt .
 7: sed -e 's/cd ga/cd ga'$2'/g' < ga.sh > ga$2.sh
 8: sed -e 's/7 3/7 3 '$1'/g' < p.txt > parameters.txt
 9: mv status.txt status$2.txt
10: rm -f ga.sh
11: rm -f p.txt
12: qsub ga$2.sh
```

Figure 4.67 - The script to install and submit an instance of the signature recognition program for execution on the computational cluster.

Figure 4.67 presents the shell script for the signature recognition program. The batch file and the directories “Code” and “Paths” are in the home directory of the user account on the host computer. The Code directory contains the signature recognition program’s executable file along with any other files needed during the execution of the program. The Paths directory contains all 100 paths files of the data set used by the experiment, the non-crossover paths data set or the crossover paths data set.

As an example, the command “ga 1 001” executes at the Linux operating system prompt, within the user account’s home directory, submitting a job to the SGE. The job creates an instance of the signature recognition program to execute on one of the cluster’s computers. The shell script ga.sh has two command line arguments represented by $1 and $2 within the script. The first command line argument ($1) is the group number for the paths processed by this instance of the signature recognition program. For the example above, the group number is 1. The second command line argument ($2) is the three digit version of the group number with leading zeros used for the single and double digit numbers. For the example above, the three digit version of the group number is 001.

The command at line 1 in the script of figure 4.67 launches the bash shell interpreter. The command at line 2 of figure 4.67 specifies to the SGE that the instance of the signature recognition program executes on a cluster computer within a bash shell. The command at line 3 in figure 4.67 creates a directory whose name has the following format. The directory name begins with the two letters “ga” and ends with the three digit number stored in the command line argument $2. For the example above, the directory name is ga001. The command at line 4 in figure 4.67 changes directories from the

current directory to the directory created by the command at line 3 in figure 4.67. The command on line 5 in figure 4.67 copies all of the files in the Code directory to the current directory. The command on line 6 in figure 4.67 copies the appropriate paths file from the Paths directory to the current directory. The copy command from line 6 uses the three digit number of the second command line argument ($2) to create the appropriate paths filename. For the example above, the appropriate paths filename is paths001.txt. The command on line 7 in figure 4.67 makes a copy of the script file ga.sh searching for the string "cd ga" within the copy of the script file. The command from line 7 appends the string "cd ga" with the three digit number of the command line argument $2. The command from line 7 uses the three digit number of the second command line argument ($2) to rename the copy of the script file with the appropriate script filename. For the example above, the appropriate script filename is ga001.sh. The command on line 8 in figure 4.67 makes a copy of the file p.txt searching for the string "7 3" within the copy of the file. The command from line 8 appends the string "7 3" with a space and the number in the command line argument $1. The command from line 8 renames the copy of the file p.txt as parameters.txt. The command at line 9 in figure 4.67 uses the three digit number of the second command line argument ($2) to rename the file status.txt with the appropriate status filename. For the example above, the appropriate status filename is status001.txt. The commands at lines 10 and 11 in figure 4.67 delete the files ga.sh and p.txt, respectively. Finally, the command at line 12 in figure 4.67 submits a batch job to the SGE in the form of the script file created by the command at line 7 in figure 4.67. The SGE is responsible for locating a computer in the cluster and submitting the batch job script to that computer for execution.

```
1: #!/bin/bash
2: cd ga001
3: ./ga
```

Figure 4.68 - An example of a batch job script file.

Figure 4.68 lists the batch job script file produced (line 7, figure 4.67) by the script file executed by the command "ga 1 001". The command at line 1 in figure 4.68 launches the bash shell interpreter on the particular cluster computer executing the job. The command at line 2 in figure 4.68 changes directories from the current directory to the directory ga001. The final command at line 3 in figure 4.68 begins execution of the signature recognition program. The next section provides a detailed description of the landscape program's main method.

4.11 The Landscape of the Signature Recognition Program Search Space

Figure 4.69 lists the code for the main method of the landscape program. Line 1 in figure 4.69 defines a constant specifying the number of agents. Line 2 in figure 4.69 defines a constant specifying the number of paths. The statements at lines 3 and 4 in figure 4.69 include, into the landscape program, the header files for the standard C++ input and output libraries. The statements at lines 5 and 6 in figure 4.69 include, into the landscape program, the header files for the user defined classes FSM and Grid. The FSM class implements the agent and the Grid class implements the grid containing the path for the landscape program.

```
 1: #define NUM_FSM   1000000
 2: #define NUM_PATHS 100
 3: #include <iostream.h>
 4: #include <fstream.h>
 5: #include "..\Common\FSM.h"
 6: #include "..\Common\Grid.h"
 7:
 8: int main()
 9: {
10:   char pathsFilename[13], outputFilename[17];
11:   int i, j, k;
12:   int gridRows, gridColumns;
13:   int numInputs, numOutputs, numStates;
14:   int maxMoveMult, fileNum;
15:   ifstream inFile;
16:   ofstream outFile;
17:   Grid *worlds[NUM_PATHS];
18:   FSM *theAgent;
19:
20:   inFile.open("parameters.txt", ios::in);
21:   inFile >> gridRows >> gridColumns;
22:   inFile >> numInputs >> numOutputs >> numStates;
23:   inFile >> maxMoveMult >> fileNum;
24:   inFile.close();
25:
26:   createPathsFilename(pathsFilename, fileNum);
27:   inFile.open(pathsFilename, ios::in);
28:   for (j=0; j<NUM_PATHS; j++)
29:   {
30:     worlds[j] = new Grid(gridRows, gridColumns);
31:     worlds[j]->addNewPathToGrid(maxMoveMult, inFile);
32:   }
33:   inFile.close();
34:
35:   createOutputFilename(outputFilename, fileNum);
36:   outFile.open(outputFilename, ios::out);
37:   FSM::setFSM(numInputs, numOutputs, numStates, 1, 1);
38:   theAgent = new FSM();
39:
40:   for (i=1; i<=NUM_FSM; i++)
41:   {
42:     theAgent->createRandomFSM();
43:     outFile << i << "\t";
44:     outFile << theAgent->reachableStates();
45:     for (j=0; j<NUM_PATHS; j++)
46:     {
47:       k = worlds[j]->computeFSMFitness(*theAgent);
```

```
48:         outFile << "\t" << k;
49:       }
50:       outFile << endl;
51:     }
52:     outFile.close();
53:
54:     return 0;
55: }
```

Figure 4.69 - The code for the program generating the signature recognition problem landscape.

The main method (lines 8 to 55, figure 4.69) is the entire landscape program. Line 20 in figure 4.69 opens the parameters file parameters.txt. Lines 21 through 23 in figure 4.69 read the following landscape parameters (section 4.1): gridRows, gridColumns, numInputs, numOutputs, numStates, maxMoveMult and fileNum from the parameters file parameters.txt. The parameters file closes at line 24 in figure 4.69.

A call (line 26, figure 4.69) to the createPathsFilename method (figure 4.61), passing pathsFilename and fileNum as parameters, creates the filename for the paths file. The paths file opens for input at line 27 in figure 4.69. The "for" loop at lines 28 through 32 in figure 4.69 controls the iteration of the variable *j* beginning with the value of 0 and increasing *j*'s value by 1 until *j* is equal to NUM_PATHS. At line 30 in figure 4.69, the j^{th} element of the array *world* is a Grid object whose constructor initializes the grid with gridRows number of rows and gridColumns number of columns. At line 31 in figure 4.69, a call to the method addNewPathToGrid of the Grid class reads the j^{th} path in the paths file and re-creates the path in the grid, the j^{th} element of the array *world*. The maximum number of moves multiplier, maxMovesMult, and the file variable inFile are parameters passed to the method addNewPathToGrid. The paths file closes at line 33 in figure 4.69.

```
void createOutputFilename(char *outputFilename,
                          int fileNum)
{
  outputFilename[0] = 'l';
  outputFilename[1] = 'a';
  outputFilename[2] = 'n';
  outputFilename[3] = 'd';
  outputFilename[4] = 's';
  outputFilename[5] = 'c';
  outputFilename[6] = 'a';
  outputFilename[7] = 'p';
  outputFilename[8] = 'e';
  outputFilename[9] = fileNum / 100 % 10 + '0';
  outputFilename[10] = fileNum / 10 % 10 + '0';
  outputFilename[11] = fileNum % 10 + '0';
  outputFilename[12] = '.';
  outputFilename[13] = 't';
  outputFilename[14] = 'x';
  outputFilename[15] = 't';
  outputFilename[16] = '\0';
}
```

Figure 4.70 - The method to create the filename for the file to contain the program results.

A call (line 35, figure 4.69) to the createOutputFilename method (figure 4.70), passing outputFilename and fileNum as parameters, creates the filename for the file containing the landscape program results. The method createOutputFilename in figure 4.70 creates the output filename in the 17 character array parameter outputFilename where the fileNum parameter specifies the unique group number for the particular copy of the landscape program. Each filename begins with the string “landscape”. The next three characters in the filename are the digits corresponding to the group number. The filename ends with the extension “.txt”. Assume fileNum is 1, the output filename is landscape001.txt. The output file opens for output at line 36 in figure 4.69.

A call (line 37, figure 4.69) to the setFSM method of the FSM class, passing numInputs, numOutputs, numStates, 1 and 1 as parameters, initializes the static data

members of the FSM class with these five values. At line 38 in figure 4.69, the variable theAgent is an FSM object whose constructor creates an instance of the FSM class for that variable.

The "for" loop at lines 40 through 51 in figure 4.69 controls the iteration of the variable *i* beginning with the value of 1 and increasing *i*'s value by 1 until *i* is greater than NUM_FSM. At line 42 in figure 4.69, a call to the FSM method createRandomFSM randomly chooses valid output and next state values to fill the respective output and next states values in the agent's finite state table. Each line in the output file contains values produced by the agent created on line 42 in figure 4.69. A tab character, '\t', separates each value on the line. The print statements at lines 43 and 44 in figure 4.69 writes, to the output file, the value of the variable *i* followed by the number of reachable states of theAgent computed by a call to the FSM method reachableStates. Nested inside the "for" loop controlling the variable *i* is the "for" loop at lines 45 to 49 in figure 4.69 which controls the iteration of the variable *j* beginning with the value of 0 and increasing *j*'s value by 1 until *j* is equal to NUM_PATHS. Inside the "for" loop controlling *j*, the print statement at line 48 in figure 4.69 writes, to the output file, the fitness value of theAgent for the j^{th} path computed at line 47 in figure 4.69 by a call to the computeFSMFitness method of the Grid class. After the "for" loop controlling *j* terminates execution, the print statement at line 50 in figure 4.69 writes a newline character to the output file to position the output file on the next line for the next possible agent. After the "for" loop controlling *i* terminates execution, the output file closes at line 52 in figure 4.69. Finally, the main method terminates by executing a return statement (line 54, figure 4.69) to

return a value of 0. The next chapter describes the algorithmic implementations for the optical character recognition problem.

Chapter 5 Optical Character Recognition Algorithmic Implementations

This chapter contains all of the C++ program code for the optical character recognition system discussed in chapter 3. The discussion of the optical character recognition system begins in the next section with the setup and control parameters for both the OCR and CDC programs. Section 5.2 provides a detailed description of the various parameters and methods for the line recognition genetic algorithm. Section 5.3 provides a detailed description of the nested loops to search the database for the font name and font size combination that closely matches the font name and font size combination of the connected component. Finally, Section 5.4 provides a detailed description of the nested loops to search the database for the character that closely matches the connected component.

5.1 Parameters for the Optical Character Recognition System

Each program uses its own copy of the parameters.txt file to contain all of the program setup and control parameters. The description of the parameters common to both programs begins in the next paragraph. The two paragraphs at the end of this section describe the parameters specific to each program. Figure 5.1 lists all of the parameters for the OCR program and figure 5.2 lists all of the parameters for the CDC program.

The parameter inFilename stores the name of the TIFF image file containing the scanned page of text. For the OCR program, the page of text is the text submitted for optical character recognition. For the CDC program, the page of text is the set of printable characters for a particular font name and font size combination to store in the

character database. The orientation parameter provides each program with the orientation, portrait or landscape, of the text in the image.

Parameter	Description	Value
inFilename	TIFF image's filename.	A string
outFilename	Text file's filename.	A string
Orientation	Page orientation.	Portrait or landscape
fontSwitch	Binary value to turn on or off the input of a font name and size.	1 or 0
fontName	Font name.	A string
fontSize	Font size.	Positive integer
maxNumGen	Maximum number of generations.	30
crossoverRate	Crossover rate.	.07
mutationRate	Mutation rate.	.03
measureSwitches	An array containing a binary value for each measure to turn on or off the use of the measure.	An array of ones and zeros

Figure 5.1 - The optical character recognition program parameters.

Parameter	Description	Value
Orientation	Page orientation.	portrait or landscape
Preprocess	Binary value to specify whether preprocessing or normal processing is done.	1 or 0
maxNumGen	Maximum number of generations.	30
crossoverRate	Crossover rate.	.07
mutationRate	Mutation rate.	.03
measureSwitches	An array containing a binary value for each measure to turn on or off the use of the measure.	An array of ones and zeros
inFilename	TIFF image's filename.	A string
fontName	Font name.	A string
fontSize	Font size.	Positive integer
keyFilename	Character key's filename	A string

Figure 5.2 - The character database creation program parameters.

It is conceivable that the line recognition genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm

terminates if it doesn't find a solution by the time the generation number equals the value of the maxNumGen parameter which is 30. The value of 30 for the maximum number of generations (maxNumGen) provides a reasonable chance for the genetic algorithm to converge. The crossover and mutation operators use the crossoverRate and mutationRate parameters, respectively, to create an offspring from two parents. The crossover and mutation rate parameter values for this genetic algorithm are 7% and 3%, respectively.

The OCR program utilizes 19 statistical measures for character recognition. An array of 19 binary values, called measureSwitches, provides the CDC and OCR programs control over the use of each measure. Each element of the array corresponds to one of the measures where a value of 1 turns on the measure and a value of 0 turns off the measure. The CDC program also uses the measureSwitches parameter because the OCR and CDC programs share a large portion of C++ code utilizing measureSwitches. For the CDC program, the value of every element in the measureSwitches array is 1 because the CDC program computes and stores, in the character's database record, every measure of each character.

This paragraph describes the parameters used solely within the OCR program. The normal execution of the OCR program's character recognition phase on a connected component is a two step process. In the first step, the OCR program searches the database for the font name and font size combination that closely fits the component. In the second step, the character recognition restricts itself to only those characters in the database having the font name and font size combination found in step one. The value of the fontSwitch parameter allows the OCR program to control the execution of step one. During some experiments, the OCR program skips the first step in order to test the

performance of the second (character recognition) step. The font name and font size recognition step executes when the fontSwitch parameter's value is 1. If fontSwitch's value is 0 then the OCR program skips the first step and uses the font name and font size combination, for the second (character recognition) step, supplied via the parameters fontName and fontSize, respectively. The parameter outFilename stores the name of the text file to hold the recognized characters.

This paragraph describes the parameters used solely within the CDC program. The CDC program receives as parameters the filename of the image file (inFilename), the font name (fontName) and font size (fontSize) of the text in the image file. The image file contains the set of all printable characters in a particular font name and font size combination. The keyFilename parameter stores the name of the text file containing the character key for the image file. Each line in the key file is the corresponding line in the image. The characters in the key file line are the same characters and in the same order as in the corresponding line in the image. The preprocess parameter is a binary value to specify whether the CDC program performs preprocessing, value of 1, or adds the characters to the database, value of 0. The next section provides a detailed description of the various parameters and methods for the line recognition genetic algorithm.

5.2 The Genetic Algorithm to Recognize the Lines in the Image

The second phase in the OCR and CDC programs uses a genetic algorithm to determine the number of lines of text in the image as well as the position of each line in the image. The next subsection describes the optical character system parameters utilized by the line recognition genetic algorithm. Subsection 5.2.2 defines and describes the

random number generator used in the line recognition genetic algorithm. Finally, subsection 5.2.3 gives a detailed description of the line recognition genetic algorithm method.

5.2.1 The Line Recognition Genetic Algorithm Parameters

Four of the optical character system parameters (described in section 5.1) are parameters used by the line recognition genetic algorithm. The genetic algorithm possibly executes for the maximum number of generations specified by the maxNumGen parameter. The crossover and mutation rate parameters control the creation of offspring from two parents during the formation of the next generation in the genetic algorithm.

The 8.5 inch side of the page of text always aligns to the x-axis of the image and the 11 inch side of the page of text always aligns to the y-axis of the image during the scan of the page. But, the lines of text in the image can run parallel to the x-axis, portrait orientation, or run parallel to the y-axis, landscape orientation. The genetic algorithm's fitness function uses the text orientation parameter specifying the text's orientation. The next subsection discusses the random number generator employed by the line recognition genetic algorithm.

5.2.2 The Line Recognition Genetic Algorithm's Random Number Generator

This line recognition genetic algorithm uses the random number generator coded by Ladd (1995). The static class called Random incorporates Ladd's (1995) random number generator. Figure 5.3 lists the declaration of the Random class making up the header file Random.h. The Random class has one static member randNumGen (line 10,

figure 5.3) of type RandDev. The class RandDev is the class written by Ladd (1995) defining his random number generator. The two methods of the Random class provide a convenient interface to Ladd's (1995) random number generator. The first method randomInt (line 6, figure 5.3) provides a means to randomly choose an integer from a range of integers. The second method randomFloat (line 7, figure 5.3) provides a means to randomly choose a floating point number from a range of floating point numbers.

```
 1: #include "RandDev.h"
 2:
 3: class Random
 4: {
 5:   public:
 6:     static int randomInt(int, int);
 7:     static float randomFloat(float, float);
 8:
 9:   private:
10:     static RandDev randNumGen;
11: };
```

Figure 5.3 - Declaration of the random number generator class.

```
 1: #include "Random.h"
 2:
 3: RandDev Random::randNumGen;
 4:
 5: int Random::randomInt(int sizeOfRange,
 6:                       int offsetOfRange)
 7: {
 8:   return ((int) (randNumGen() * sizeOfRange +
 9:                  offsetOfRange));
10: }
11:
12: float Random::randomFloat(float sizeOfRange,
13:                           float offsetOfRange)
14: {
15:   return (randNumGen() * sizeOfRange + offsetOfRange);
16: }
```

Figure 5.4 - Definition of the methods of the random number generator class.

Figure 5.4 lists the definition of the Random class making up the source code file Random.cpp. Line 1 in figure 5.4 includes the header file Random.h in the source file. The static member randNumGen (line 3, figure 5.4) creates an instance of data type RandDev initialized by RandDev's constructor.

The method randomInt (lines 5 to 10, figure 5.4) randomly chooses an integer from a particular range of integers. The first parameter of the method, sizeOfRange, defines the size of the range of integers. Ladd (1995) defined the () operator in his RandDev class to randomly generate a floating point value that is greater than or equal to 0 and less than 1. A call to this () operator takes the form of randNumGen() in the Random class. At line 8 in figure 5.4, the call to randNumGen() multiplied by the parameter sizeOfRange produces a randomly chosen value that is greater than or equal to 0 and less than sizeOfRange. For example, if sizeOfRange is 4, the randomly chosen value is greater than or equal to 0 and less than 4. The second parameter of the method, offsetOfRange, specifies an offset value to allow the randomly chosen value to come from any possible range of integers. The randomInt method adds offsetOfRange to the result of multiplying randNumGen() by sizeOfRange as seen in lines 8 and 9 in figure 5.4. For example, if sizeOfRange is 4 and offsetOfRange is 6, the randomly chosen value is greater than or equal to 6 and less than 10. Since the randomly chosen value is a floating point value, a cast to int is necessary because the method randomInt returns a randomly chosen integer.

The method randomFloat (lines 12 to 16, figure 5.4) randomly chooses a floating point number from a particular range of floating point numbers. The first parameter of the method, sizeOfRange, defines the size of the range of floating point numbers. The

second parameter of the method, offsetOfRange, specifies an offset value to allow the randomly chosen value to come from any possible range of floating point numbers. The only difference between the definition of randomFloat and randomInt is the cast to int not needed by randomFloat before returning the randomly chosen floating point number. The next subsection describes the line recognition genetic algorithm method.

5.2.3 The Line Recognition Genetic Algorithm

The line recognition genetic algorithm evolves a population of lines, searching for a line whose slope closely matches the slope of the lines of text in the image. The method geneticAlgorithm in figure 5.5 executes the line recognition genetic algorithm for the page of text in the image.

```
 1: void geneticAlgorithm(int maxNumGen,
 2:                       Population *theGALines,
 3:                       Components *theComps)
 4: {
 5:   int i;
 6:
 7:   theGALines->initializePopulation();
 8:   theGALines->computeEachFitness(theComps);
 9:   for (i=1; i<=maxNumGen; i++)
10:   {
11:     theGALines->createNewPop();
12:     theGALines->computeEachFitness(theComps);
13:   }
14: }
```

Figure 5.5 - The code for the line recognition genetic algorithm method.

The parameter maxNumGen represents the maximum number of generations for execution of the genetic algorithm. The parameter theGALines is a pointer to a Population object containing the entire population of lines. The parameter theComps is a

pointer to a Components object containing all of the connected components in the image as well as the sets gaComponents and nonGAComponents.

The genetic algorithm begins at line 7 in figure 5.5 by randomly generating the initial population of lines via a call to the initializePopulation method of the Population class. Next, a call (line 8, figure 5.5) to the computeEachFitness method of the Population class, passing theComps as a parameter, computes the fitness value for each member of the population. The "for" loop at lines 9 through 13 in figure 5.5 generates generation after generation of lines until the genetic algorithm reaches the maximum number of generations. The variable *i* stores the current generation number and initializes to generation 1 at line 9 in figure 5.5. A call (line 11, figure 5.5) to the createNewPop method of the Population class creates the next generation of lines. The next generation of lines becomes the population for the current generation. The genetic algorithm computes the fitness value of each member of the population via a call (line 12, figure 5.5) to the computeEachFitness method of the Population class passing theComps as a parameter. The "for" loop (the genetic algorithm) terminates (line 9, figure 5.5) when *i* is greater than maxNumGen, the maximum number of generations. Upon termination of the line recognition genetic algorithm, the fitness value of the fittest individual provides the number of lines in the image. The next section provides a detailed description of the nested loops to search the database for the font name and font size combination that closely matches the font name and font size combination of the connected component.

5.3 The Font Name and Font Size Recognition Step

The first step of the connected component's optical character recognition searches the database for the font name and font size combination that closely matches the font name and font size combination of the connected component. The search executes SQL Select statements on the Properties table using a range of values for each statistical measure. Figure 5.6 lists the C++ code to repeatedly generate and execute the Select statements on the Properties table. The "for" loop at lines 6 through 15 in figure 5.6 controls the variable *percent* storing the percentage. The variable perInc is *percent*'s initial value (line 6, figure 5.6) as well as the value to increment *percent* (line 8, figure 5.6) at each iteration of the loop. This inner "for" loop terminates (line 7, figure 5.6) when the current SQL Select statement returns at least one record or when *percent* is greater than the value of the variable perFinal.

```
 1: recCount=0;
 2: for (perInc=PERCENT_INCREMENT, perFinal=PERCENT_FINAL;
 3:      recCount<=0;
 4:      perInc*=PERCENT_FACTOR, perFinal*=PERCENT_FACTOR)
 5: {
 6:   for (percent=perInc;
 7:        recCount<=0 && percent<=perFinal;
 8:        percent+=perInc)
 9:   {
10:     sql.emptyString();
11:     createFontSelectQuery(percent, sql);
12:     result = &(db.ExecuteQuery(sql.getString()));
13:
14:     for (recCount=0; result->MoveNext(); recCount++) ;
15:   }
16: }
```

Figure 5.6 - The code to search for a connected component's closest matching font name and font size combination.

The outer "for" loop (lines 2 to 16, figure 5.6) controls the variables perInc and perFinal which specify the initial and final values, respectively, for the inner "for" loop's control variable *percent*. The initial value of perInc is the value (5%) of the program constant PERCENT_INCREMENT and the initial value of perFinal (100%) is the value of the program constant PERCENT_FINAL. For the first execution of the inner "for" loop, *percent*'s initial value is 5% incrementing by 5% for each iteration of the loop. Unless an SQL Select statement returns at least one record, the inner loop terminates when *percent* is greater than 100%. The outer loop executes another iteration unless an SQL Select statement returns at least one record. The value (10) of the program constant PERCENT_FACTOR multiplies the outer loop control variables perInc and perFinal. For example, during the second execution of the inner "for" loop, *percent*'s initial value is 50% incrementing by 50% for each iteration of the loop. Unless an SQL Select statement returns at least one record, the inner loop terminates when percent is greater than 1000%. Eventually, the outer loop terminates when an SQL Select statement produces at least one record. The next section provides a detailed description of the nested loops to search the database for the character that closely matches the connected component.

5.4 The Character Recognition Step

The second step of the connected component's optical character recognition searches the database for the character that closely matches the connected component with the search restricted to the font name and font size combination either found in the first step or supplied to the OCR program. The search executes SQL Select statements on

the Characters table using a range of values for each statistical measure. Figure 5.7 lists the C++ code to repeatedly generate and execute the Select statements on the Characters table. The "for" loop at lines 6 through 15 in figure 5.7 controls the variable *percent* storing the percentage. The variable perInc is *percent*'s initial value (line 6, figure 5.7) as well as the value to increment *percent* (line 8, figure 5.7) at each iteration of the loop. This inner "for" loop terminates (line 7, figure 5.7) when the current SQL Select statement returns at least one record or when *percent* is greater than the value of the variable perFinal.

```
 1: recCount=0;
 2: for (perInc=PERCENT_INCREMENT, perFinal=PERCENT_FINAL;
 3:      recCount<=0;
 4:      perInc*=PERCENT_FACTOR, perFinal*=PERCENT_FACTOR)
 5: {
 6:   for (percent=perInc;
 7:        recCount<=0 && percent<=perFinal;
 8:        percent+=perInc)
 9:   {
10:     sql.emptyString();
11:     createCharSelectQuery(percent, sql);
12:     result = &(db.ExecuteQuery(sql.getString()));
13:
14:     for (recCount=0; result->MoveNext(); recCount++) ;
15:   }
16: }
```

Figure 5.7 - The code to search for a connected component's closest matching character.

The outer "for" loop (lines 2 to 16, figure 5.7) controls the variables perInc and perFinal which specify the initial and final values, respectively, for the inner "for" loop's control variable *percent*. The initial value of perInc is the value (5%) of the program constant PERCENT_INCREMENT and the initial value of perFinal (100%) is the value of the program constant PERCENT_FINAL. For the first execution of the inner "for"

loop, *percent*'s initial value is 5% incrementing by 5% for each iteration of the loop. Unless an SQL Select statement returns at least one record, the inner loop terminates when *percent* is greater than 100%. The outer loop executes another iteration unless an SQL Select statement returns at least one record. The value (10) of the program constant PERCENT_FACTOR multiplies the outer loop control variables perInc and perFinal. For example, during the second execution of the inner "for" loop, *percent*'s initial value is 50% incrementing by 50% for each iteration of the loop. Unless an SQL Select statement returns at least one record, the inner loop terminates when percent is greater than 1000%. Eventually, the outer loop terminates when an SQL Select statement produces at least one record. The next chapter describes the experiments and results from the two problems in this dissertation.

Chapter 6 Experimentation and Results

Chapter 2 presents the description of the signature recognition problem and the solution employed in this dissertation. Section 6.1 details the discussion of the five experiments of the signature recognition problem. Chapter 3 provides the description of the moment based optical character recognition problem and the solution employed in this dissertation. Section 6.2 details the discussion of the experiments of the moment based optical character recognition problem.

6.1 Signature Recognition Experiments

Recall from section 2.2 in chapter 2, there are two sets of paths to test the signature recognition program. The first set of 10,000 paths are the non-crossover paths, no part of a path can crossover any other part of the path. The next subsection, 6.1.1, discusses the properties of the non-crossover paths. The second set of 10,000 paths are the crossover paths, any part of a path can crossover any other part of the path. Subsection 6.1.2 discusses the properties of the crossover paths.

Recall from subsection 4.7.1 in chapter 4, the signature recognition genetic algorithm executes until satisfying one of the two termination conditions. An execution of the genetic algorithm is a run. The first termination condition is true when at least one agent in the current population has perfect fitness. An agent's fitness value is perfect when it is equal to the path's length; in other words, the agent consumed the entire path. The signature recognition genetic algorithm writes all agents in the current population with perfect fitness to the output file along with the path, run and generation numbers. This type of run is a successful run of the genetic algorithm.

The second termination condition is true when the genetic algorithm's final generation contains no agents with perfect fitness. Depending on the experiment, the signature recognition genetic algorithm writes nothing to the output file or writes the agent with the highest fitness value to the output file along with the path number, the run number, and a generation number one higher than the maximum generation number. This type of run is an unsuccessful run of the genetic algorithm.

Five experiments executed the signature recognition program. The first three experiments utilized the set of non-crossover paths, wrote nothing to the output file for the second termination condition, and used the fitness function computeFSMFitness (section 2.3 in chapter 2) in which the order of path consumption is immaterial. Subsection 6.1.3 discusses the results of the first experiment, subsection 6.1.4 discusses the results of the second experiment, and subsection 6.1.5 discusses the results of the third experiment.

The fourth experiment utilized the set of crossover paths, wrote nothing to the output file for the second termination condition, and used the fitness function computeFSMFitness (section 2.3 in chapter 2) in which the order of path consumption is immaterial. Subsection 6.1.6 discusses the results of the fourth experiment. The fifth experiment utilized the set of crossover paths. For the second termination condition, the fifth experiment wrote the agent with the highest fitness value to the output file along with the path number, the run number, and a generation number one higher than the maximum generation number. Finally, the fifth experiment used the fitness function computeFSMPCFitness (section 2.3 in chapter 2) in which the order of path consumption is important. The fifth experiment seeks agents that consume a path in a partial

contiguous order of which full contiguous order is a special case. Recall that a full contiguous order means the next path square the agent consumes is immediately next to the previous path square consumed and that a partial contiguous order means the agent consumes segments of the path in a full contiguous order but consumes the path squares between segments in a non-contiguous order. Subsection 6.1.7 discusses the results of the fifth experiment.

6.1.1 Non-Crossover Paths

The examination of the non-crossover paths begins in the next subsection with an analysis of the path's length. Subsection 6.1.1.2 analyzes the path's winding number, the number of 90 degree turns made by the path. Subsection 6.1.1.3 investigates the landscape of agents for the non-crossover paths. Finally, subsection 6.1.1.4 displays the non-crossover paths, from experiments one through three, with the worst performance.

6.1.1.1 Path Lengths

Retrieve the length of each non-crossover path from the files storing the non-crossover paths. The smallest path length is 13 and the largest path length is 2644. Set the entire path length range as 1 to 2800 divided up into 14 smaller ranges, each 200 in size. Count all the paths whose length falls within each range. In addition, compute the average path length for each range. The chart in figure 6.1 illustrates the range counts for the non-crossover paths with the path length ranges along the x-axis and the path count along the y-axis. The chart of the path counts appears to form a bell curve. For the non-crossover paths, the path creation program does not randomly choose the length of a path

at the beginning of the path creation but is a result of the random choices of each leg's length and direction until path creation terminates. Though, a significant number of paths cluster within the 201 to 1200 path length range, there are a large number of paths whose length is outside of that range. The table in figure 6.2 provides the average path length for each path length range. An examination of each range's average reveals the value is near the mid point of the range. It is possible that the paths, according to their length, evenly distribute within the range. The next subsection examines the path's winding number.

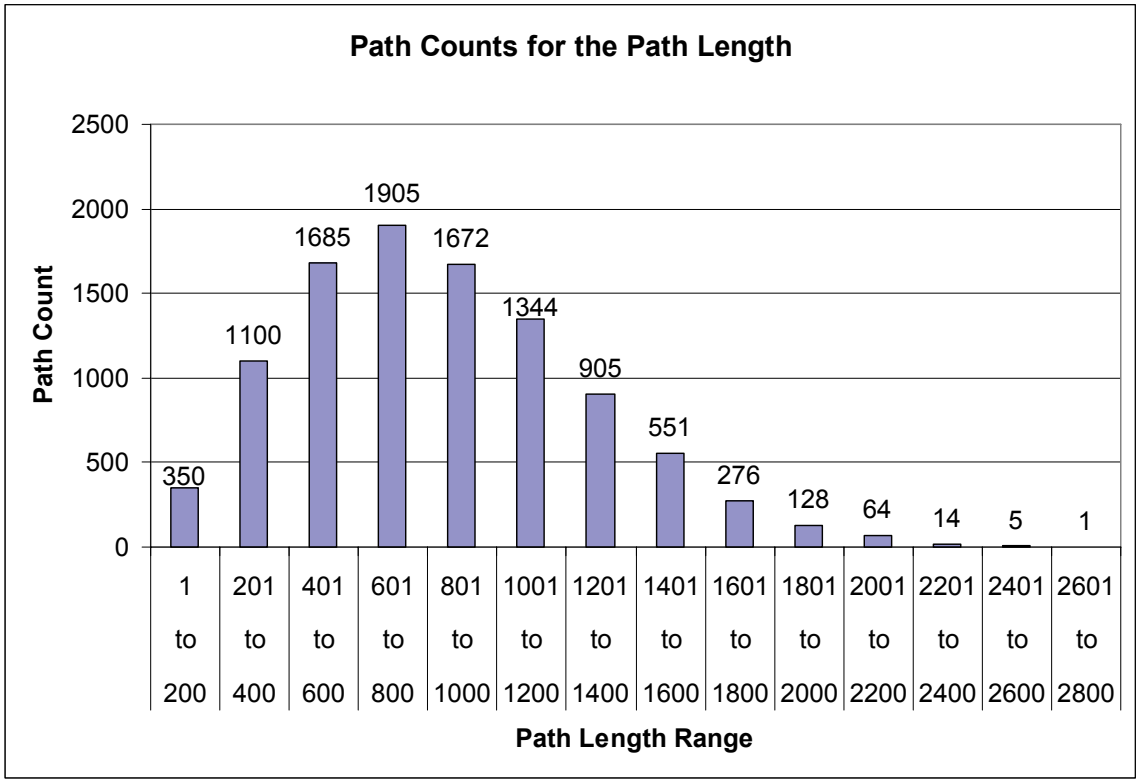


Figure 6.1 - Non-Crossover path counts for the path length.

Path Length Range	Average Path Length for the Range
1 to 200	139.27
201 to 400	312.52
401 to 600	503.06
601 to 800	702.39
801 to 1000	900.43
1001 to 1200	1092.78
1201 to 1400	1293.03
1401 to 1600	1486.85
1601 to 1800	1684.86
1801 to 2000	1887.92
2001 to 2200	2093.50
2201 to 2400	2296.93
2401 to 2600	2491.20
2601 to 2800	2644.00

Figure 6.2 - The average non-crossover path length for each path length range.

6.1.1.2 Winding Numbers

A non-crossover path's winding number is the number of 90 degree turns made by the path. The first two legs of the path form the first turn. Each remaining leg of the path forms the next turn of the path. Therefore, the non-crossover path's winding number is one less than the number of legs of the path. Recall from section 2.2 in chapter 2, the path creation program stores each non-crossover path in a file, on its own line, as a sequence of integers. Starting with the seventh and eighth numbers on the line but not the last pair, each pair represents a leg of the path. Hence, computation of the non-crossover path's winding number utilizes the path's line in the file storing the path. The smallest winding number is 2 and the largest winding number is 84. The entire winding number range is 1 to 90 divided up into 9 smaller ranges, each 10 in size. Count all of the

paths whose winding number falls within each range. In addition, compute the average winding number for each range.

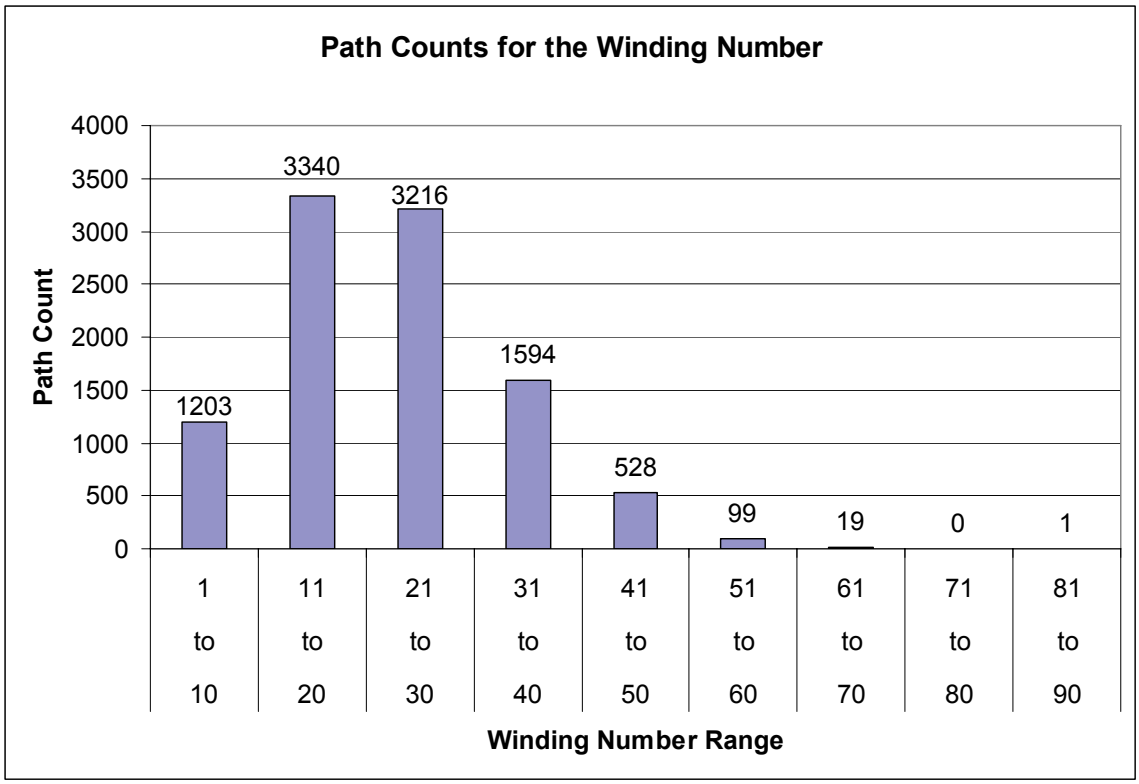


Figure 6.3 - Non-Crossover path counts for the winding number.

Winding Number Range	Average Winding Number for the Range
1 to 10	7.46
11 to 20	15.91
21 to 30	25.14
31 to 40	34.61
41 to 50	44.27
51 to 60	54.06
61 to 70	64.68
71 to 80	0.00
81 to 90	84.00

Figure 6.4 - The average non-crossover path winding number for each winding number range.

The chart in figure 6.3 illustrates the range counts for the non-crossover paths with the winding number ranges along the x-axis and the path count along the y-axis. The chart of the path counts appears to form a bell curve. For the non-crossover paths, the path creation program does not randomly choose the length of a path at the beginning of the path creation but is a result of the random choices of each leg's length and direction until path creation terminates. Though, a significant number of paths cluster within the 11 to 30 path winding number range, there are a large number of paths whose winding number is outside of that range. The table in figure 6.4 provides the average winding number for each winding number range. An examination of each range's average reveals the value is near the mid point of the range. It is possible that the paths, according to their winding number, evenly distribute within the range. The next subsection investigates the landscape of agents for the non-crossover paths.

6.1.1.3 A Landscape of Agents for the Non-Crossover Paths

To summarize the description in section 2.7 of chapter 2, the landscape program produces data about a very large group of randomly generated agents applied to the set of 10,000 non-crossover paths. Recall that the path creation program split the 10,000 paths into 100 files, each with 100 paths. The first paths file is paths001.txt and the last paths file is paths100.txt. Therefore, 100 instances of the landscape program execute in parallel, each on a different paths file. Each landscape program instance produces an output file. The landscape program instance working on the first paths file, paths001.txt, produces the output file named landscape001.txt. The first output file is landscape001.txt and the last output file is landscape100.txt.

The landscape program creates an agent by randomly choosing valid output and next state values to fill the output and next states values in the agent's finite state table. The landscape program computes the number of reachable states for the agent and writes that number to the program's output file. Using the fitness function computeFSMFitness, the landscape program computes the agent's fitness value for each path in the paths file with each fitness value written to the output file. The landscape program randomly creates 1,000,000 agents, computing and writing the values of each agent to the output file as described above.

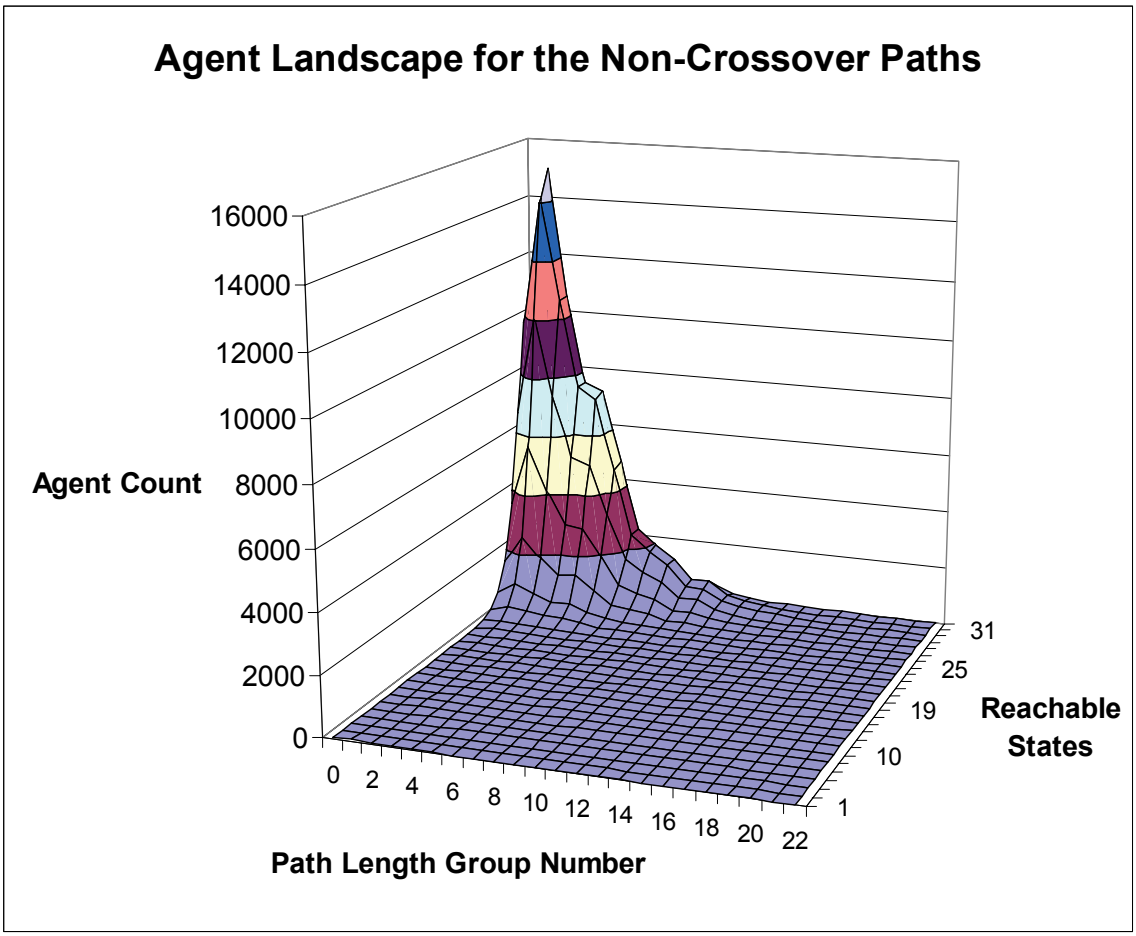


Figure 6.5 - Front view of the agent landscape of the non-crossover paths.

A separate program, landscape_stats, processes the results produced by the landscape program. The landscape_stats program utilizes a two-dimensional array, called *statistics*, to store the values for the chart seen in figures 6.5 and 6.6. There are 32 rows and 27 columns in the array *statistics*. The rows represent the number of reachable states in an agent and the columns represent the path length group number. The path length group number is equal to the path's length modulo 100. For example, all paths whose length is between 0 and 99 have a path length group number of 0. The landscape_stats program initializes each entry in the array *statistics* to 0.

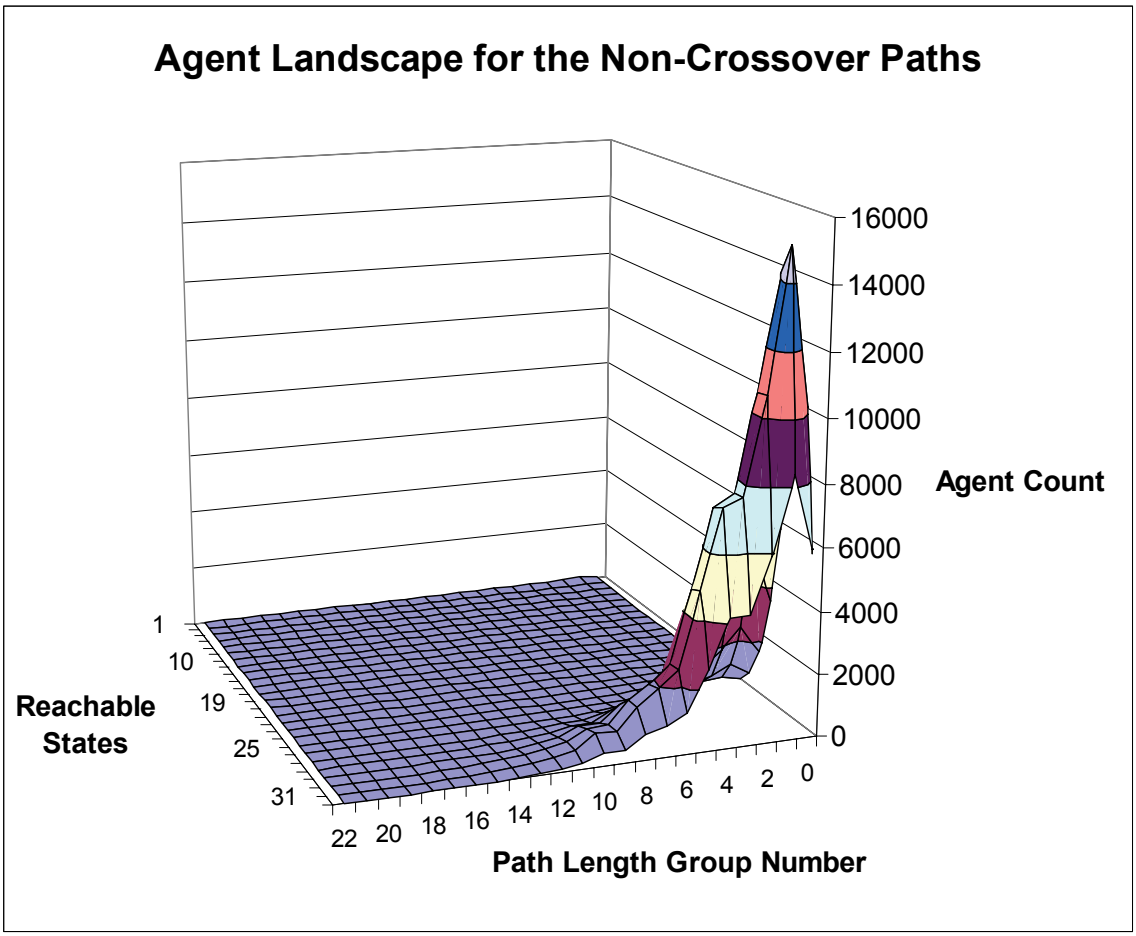


Figure 6.6 - Back view of the agent landscape of the non-crossover paths.

The landscape_stats program processes each landscape output file one at a time. The landscape_stats program opens the output file's corresponding paths file and retrieves the path length for all 100 paths. Next, the landscape_stats program opens the output file. For each agent in the output file, the landscape_stats program performs the following 2 steps. Step 1: retrieve the agent's number of reachable states and store it in the variable rs. Step 2: read and process each fitness value. The agent's i^{th} fitness value corresponds to the i^{th} path in the paths file. Compute the path length group number for the i^{th} path and store the group number in the variable plg. Compare the i^{th} fitness value to the path length of the i^{th} path. If the fitness value is equal to the path length, the agent consumed the entire path, then increment by one the element in the array *statistics* at row rs and column plg.

Once the landscape_stats program processes all of the landscape output files, each element in the *statistics* array contains a count of all the agents generated by the landscape program with the number of reachable states rs and perfect fitness for a path within the path length group number plg. The number of reachable states range from 1 to 32. The path lengths of all 10,000 non-crossover paths range from 13 to 2644. Therefore, the path length group numbers range from 0 to 26.

All the elements in the columns of the array *statistics* corresponding to the path length group numbers from 23 to 26 have counts of 0. Therefore, the chart in figures 6.5 and 6.6 does not contain these columns. The chart in figures 6.5 and 6.6 is a 3-D surface chart created within Microsoft Excel. The path length group number is along the x-axis, the number of reachable states is along the y-axis, and the agent count is along the z-axis. The figures 6.5 and 6.6 provide two different views of the chart to give two different

angles of the chart's peak. The peak spans the number of reachable states from 21 to 32 and the path length group number from 0 to about 16. The path length group number range represents paths whose length ranges from 1 to 1699. These results contain common sense. The number of reachable states from 21 to 32 provides the agent the ability to implement a large number of behaviors. The path lengths from 1 to 1699 represent short to mid-length paths which require less consumption than the longer length paths. The chart's peak exists at the number of reachable states of 32 and the path length group number of 1, paths whose length run from 100 to 199. The next subsection discusses a sample of non-crossover paths.

6.1.1.4 A Sample of Non-Crossover Paths

A review of the results of the non-crossover paths from experiments 1 through 3 discovered the paths with the least number of successful runs. The table in figure 6.7 lists the top three non-crossover paths, from experiment 1, with the least number of successful runs. The table in figure 6.8 lists the top four non-crossover paths, from experiment 2, with the least number of successful runs. The table in figure 6.9 lists the top four non-crossover paths, from experiment 3, with the least number of successful runs.

Path Number	Path Length	Number of Successful Runs	Winding Number
1611	2187	40	66
4417	2089	43	62
7473	1302	44	56

Figure 6.7 - Experiment 1's top three paths having the least number of successful runs.

In each table, each path entry lists its path number, path length, the path's number of successful runs, and the path's winding count. Recall from subsection 6.1.1.1, the smallest non-crossover path length is 13 and the largest non-crossover path length is 2644. Recall from subsection 6.1.1.2, the smallest winding number is 2 and the largest winding number is 84. All but one of the non-crossover paths in figures 6.7 through 6.9 has a path length greater than 2000. Though the path length of non-crossover path 7473 is much less than the path lengths of the other paths in figures 6.7 through 6.9; path 7473's winding number is close in value to all of the other paths in figures 6.7 through 6.9.

Path Number	Path Length	Number of Successful Runs	Winding Number
1611	2187	39	66
5809	2494	63	68
1373	2070	64	60
4417	2089	64	62

Figure 6.8 - Experiment 2's top four paths having the least number of successful runs.

Path Number	Path Length	Number of Successful Runs	Winding Number
5809	2494	49	68
531	2117	55	54
3960	2147	56	58
1611	2187	58	66

Figure 6.9 - Experiment 3's top four paths having the least number of successful runs.

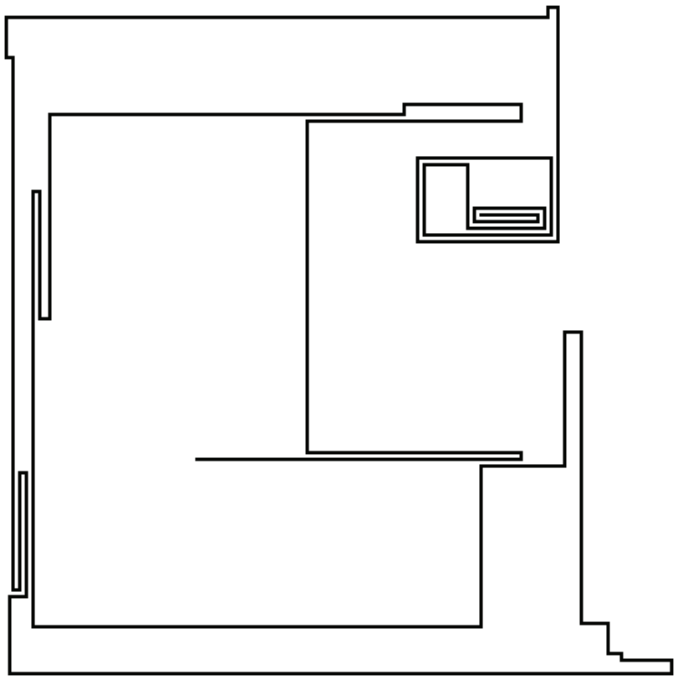


Figure 6.10 - Path 531, length: 2117, number of successful runs: 55, winding number: 54.

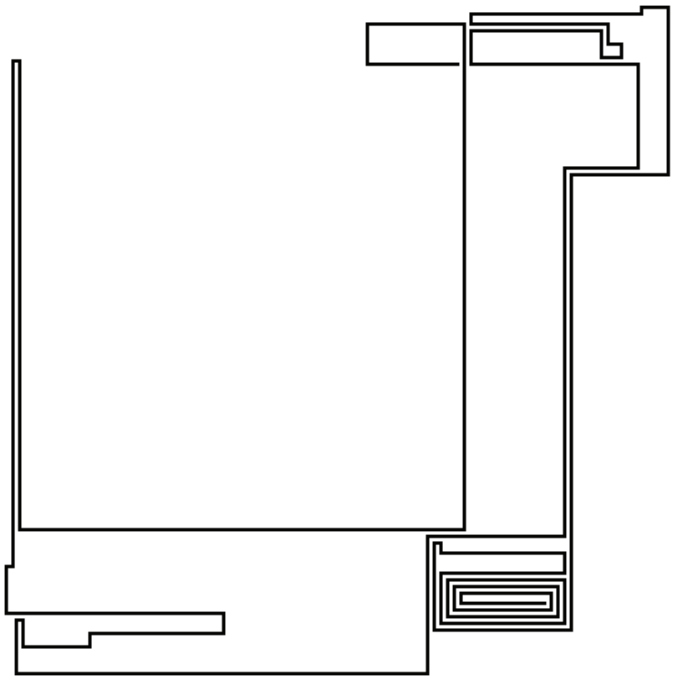


Figure 6.11 - Path 1373, length 2070, number of successful runs 64, winding number 60.

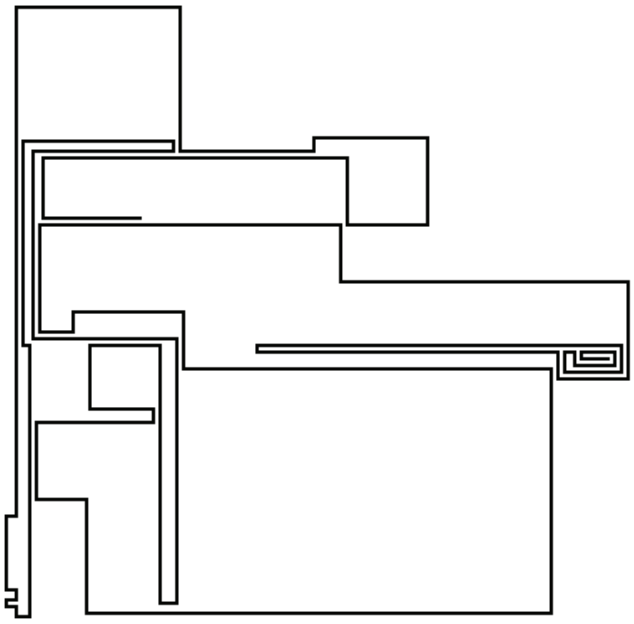


Figure 6.12 - Path 1611, length 2187, number of successful runs 39, winding number 66.

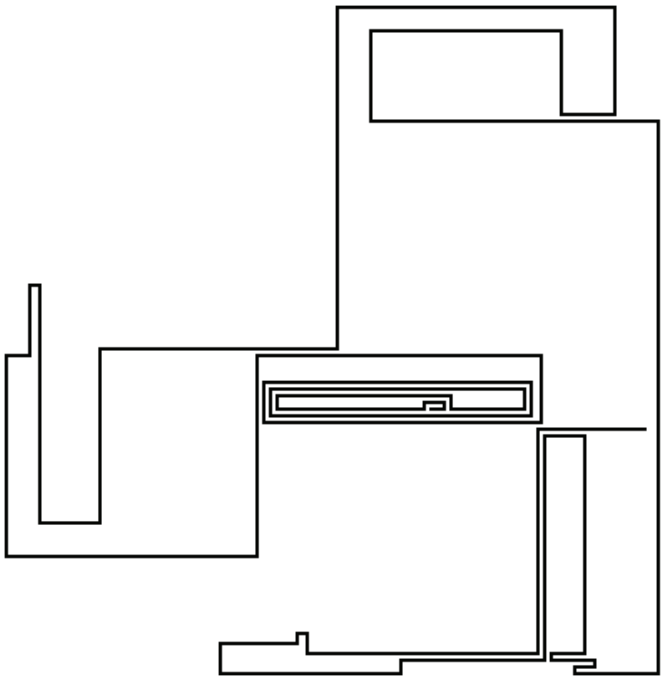


Figure 6.13 - Path 3960, length 2147, number of successful runs 56, winding number 58.

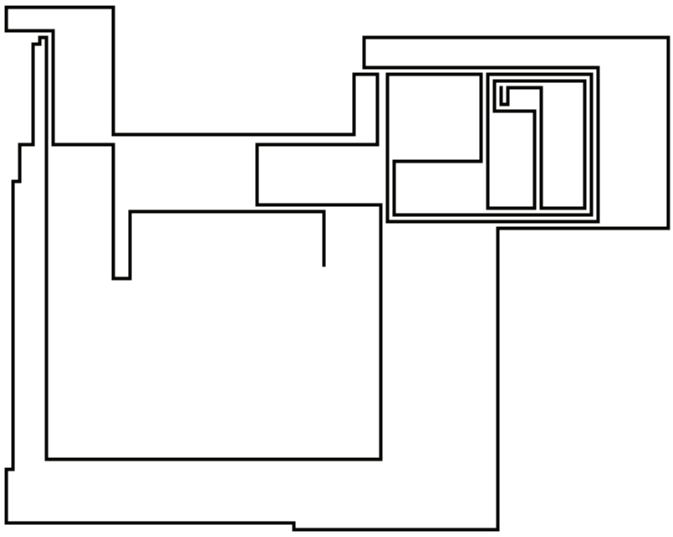


Figure 6.14 - Path 4417, length 2089, number of successful runs 43, winding number 62.

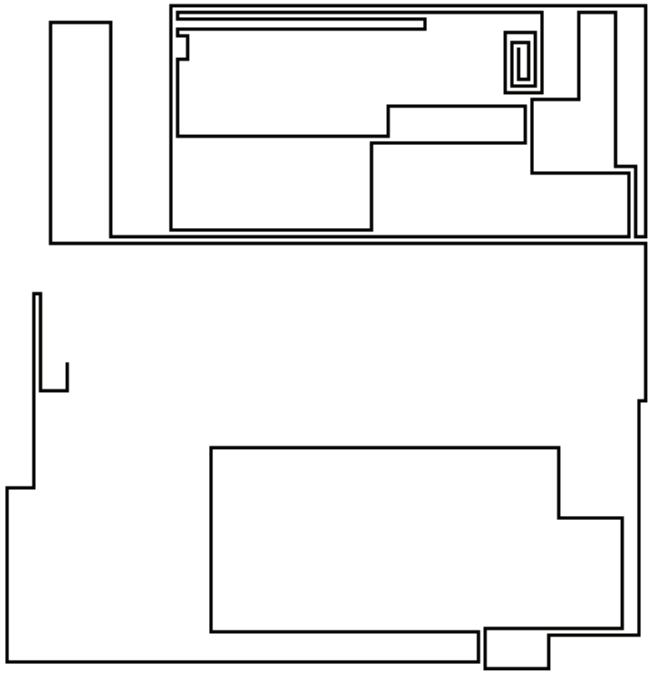


Figure 6.15 - Path 5809, length 2494, number of successful runs 49, winding number 68.

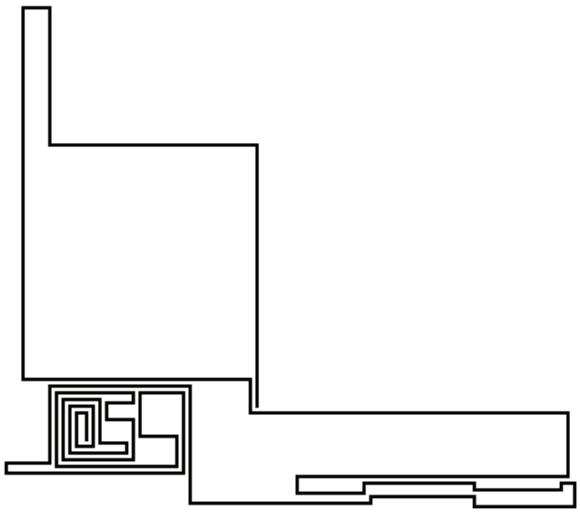


Figure 6.16 - Path 7473, length 1302, number of successful runs 44, winding number 56.

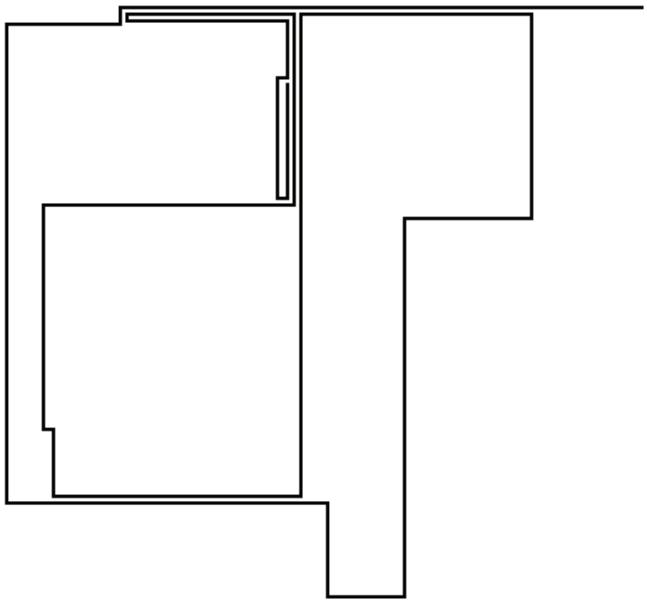


Figure 6.17 - Path 2233, length 1400, number of successful runs 100, winding number 25.

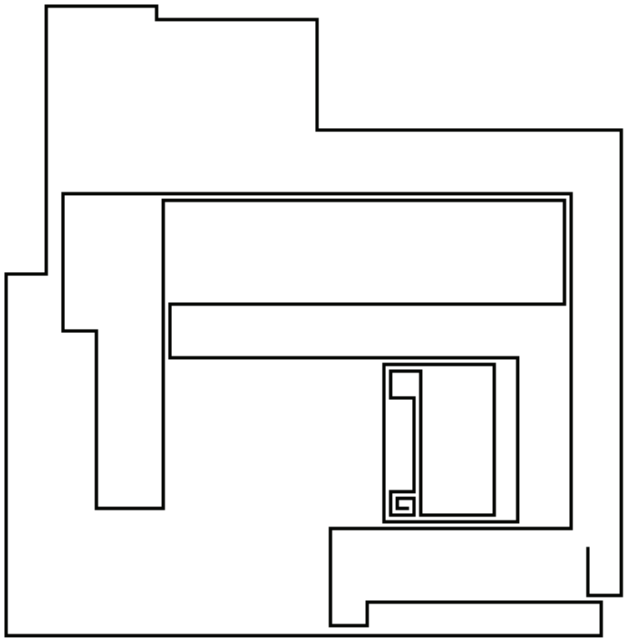


Figure 6.18 - Path 3269, length 2175, number of successful runs 100, winding number 47.

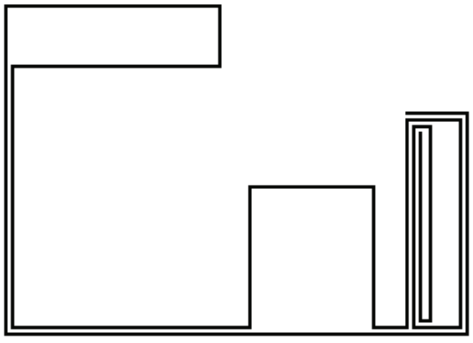


Figure 6.19 - Path 4978, length 1081, number of successful runs 100, winding number 21.

Figures 6.10 through 6.16 display an image of each of the non-crossover paths in figures 6.7 through 6.9. Figure 6.10 is an image of non-crossover path 531. Figure 6.11 is an image of non-crossover path 1373. Figure 6.12 is an image of non-crossover path 1611. Figure 6.13 is an image of non-crossover path 3960. Figure 6.14 is an image of non-crossover path 4417. Figure 6.15 is an image of non-crossover path 5809. Figure 6.16 is an image of non-crossover path 7473. In contrast to these seven paths, figures 6.17 through 6.19 display three non-crossover paths with the maximum possible number of successful runs, 100. Figure 6.17 is an image of non-crossover path 2233 with path length 1400 and winding number 25. Figure 6.18 is an image of non-crossover path 3269 with path length 2175 and winding number 47. Figure 6.19 is an image of non-crossover path 4978 with path length 1081 and winding number 21. The path lengths of these three very successful non-crossover paths are close to the lengths of the seven least successful non-crossover paths but the winding number of these three paths is a lot less than the seven paths. The winding number is the true measure of the path's difficulty. The seven least successful paths have a more complex geometry than the three very successful paths making the seven paths difficult objects for the agents to follow and consume. The winding number is the true measure of the path's difficulty. The next subsection discusses the properties of the set of crossover paths.

6.1.2 Crossover Paths

The examination of the crossover paths begins in the next subsection with an analysis of the path's length. Subsection 6.1.2.2 analyzes the path's winding number, the

number of 90 degree turns made by the path. Finally, subsection 6.1.2.3 displays the crossover paths, from experiments four and five, with the worst performance.

6.1.2.1 Path Lengths

Retrieve the length of each crossover path from the files storing the crossover paths. Just as for the non-crossover paths, the smallest crossover path length is 13 and the largest crossover path length is 2644. Set the entire path length range as 1 to 2800 divided up into 14 smaller ranges, each 200 in size. Count all the paths whose length falls within each range. In addition, compute the average path length for each range.

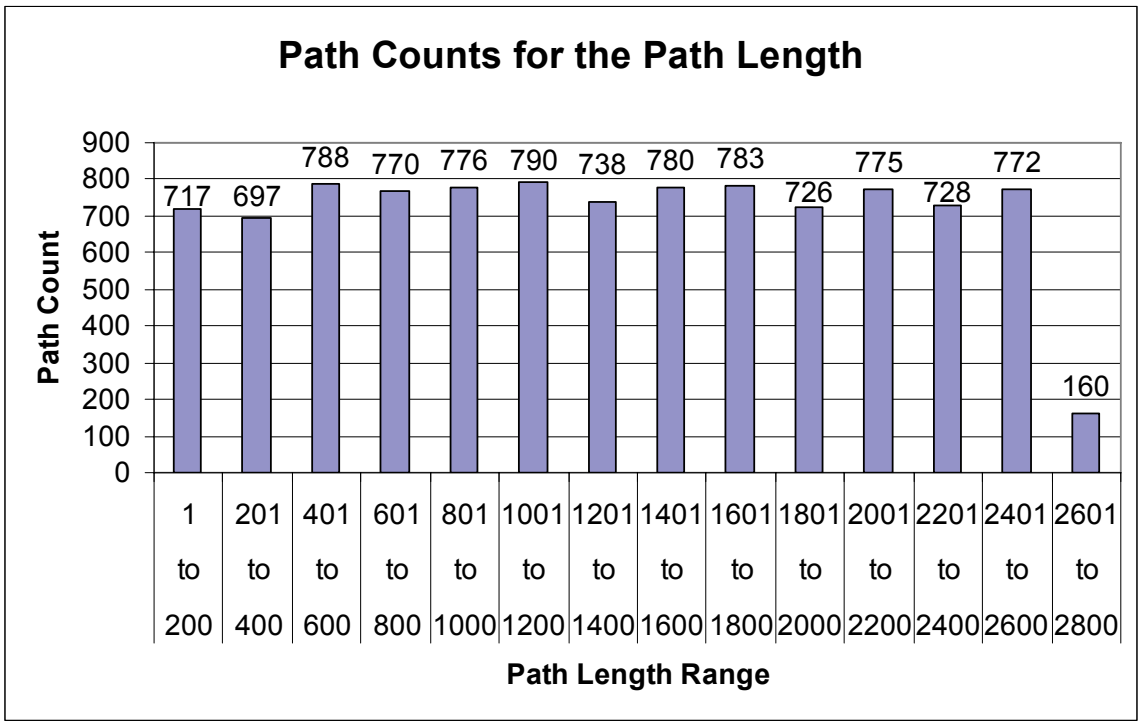


Figure 6.20 - Crossover path counts for the path length.

The chart in figure 6.20 illustrates the range counts for the crossover paths with the path length ranges along the x-axis and the path count along the y-axis. A look at the

chart shows that the counts of each range are remarkably similar. Unlike the non-crossover paths, the path creation program randomly chooses the length of a crossover path at the beginning of the path's creation. The table in figure 6.21 provides the average path length for each path length range. An examination of each range's average reveals the value is near the mid point of the range. It is possible that the paths, according to their length, evenly distribute within the range. The next subsection examines the path's winding number.

Path Length Range	Average Path Length for the Range
1 to 200	106.33
201 to 400	300.78
401 to 600	499.16
601 to 800	700.00
801 to 1000	899.58
1001 to 1200	1099.81
1201 to 1400	1301.02
1401 to 1600	1500.05
1601 to 1800	1701.54
1801 to 2000	1904.29
2001 to 2200	2099.83
2201 to 2400	2300.02
2401 to 2600	2499.30
2601 to 2800	2622.01

Figure 6.21 - The average crossover path length for each path length range.

6.1.2.2 Winding Numbers

A crossover path's winding number is the number of 90 degree turns made by the path. The first two legs of the path form the first turn. Each remaining leg of the path forms the next turn of the path. Therefore, the crossover path's winding number is one less than the number of legs of the path. Recall from section 2.2 in chapter 2, the path

creation program stores each crossover path in a file, on its own line, as a sequence of integers. Starting with the seventh and eighth numbers on the line but not the last pair, each pair represents a leg of the path. Hence, computation of the crossover path's winding number utilizes the path's line in the file storing the path. The smallest winding number is 0 and the largest winding number is 65. The entire winding number range is 0 to 69 divided up into 7 smaller ranges, each 10 in size. Count all of the paths whose winding number falls within each range. In addition, compute the average winding number for each range.

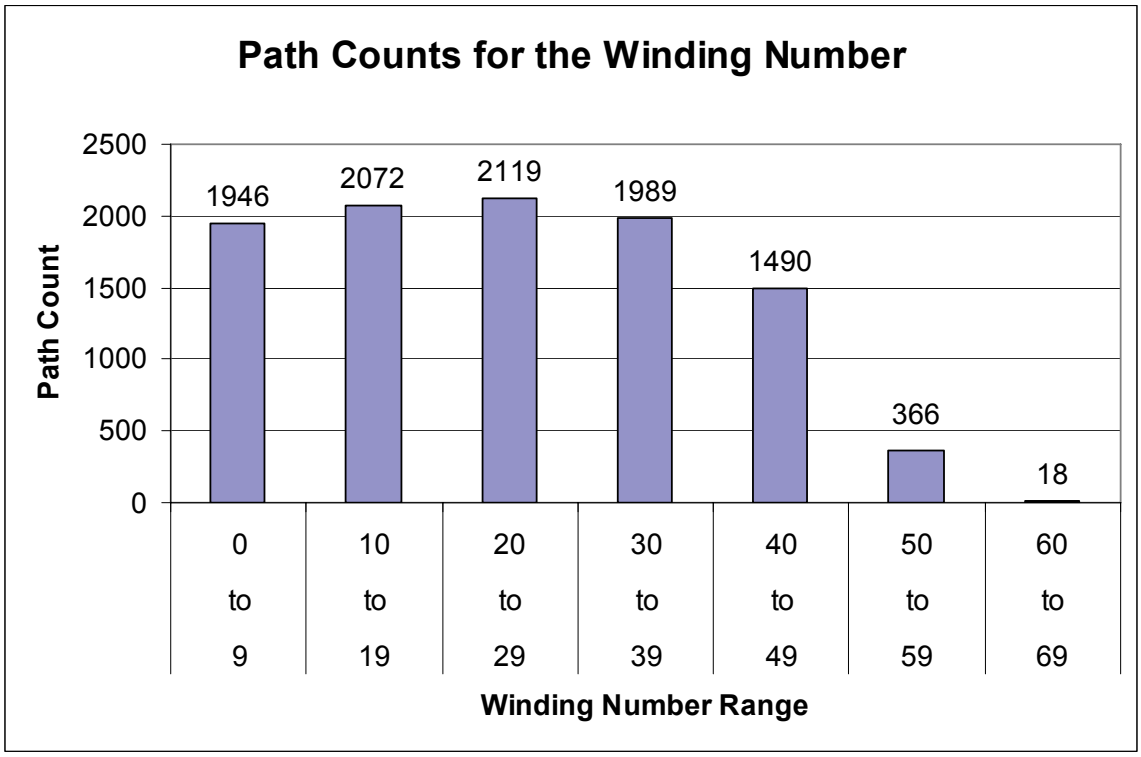


Figure 6.22 - Crossover path counts for the winding number.

The chart in figure 6.22 illustrates the range counts for the crossover paths with the winding number ranges along the x-axis and the path count along the y-axis. A look at the chart shows that the counts of the ranges from 0 to 39 are remarkably similar with

the counts tapering off for the remaining three ranges. Unlike the non-crossover paths, the path creation program randomly chooses the length of a crossover path at the beginning of the path's creation. Though, a significant number of paths cluster within the 0 to 39 path winding number range, there are a large number of paths whose winding number is outside of that range. The table in figure 6.23 provides the average winding number for each winding number range. An examination of each range's average reveals the value is near the mid point of the range. It is possible that the paths, according to their winding number, evenly distribute within the range. The next subsection discusses a sample of crossover paths.

Winding Number Range	Average Winding Number for the Range
0 to 9	4.64
10 to 19	14.54
20 to 29	24.46
30 to 39	34.49
40 to 49	43.84
50 to 59	52.89
60 to 69	62.17

Figure 6.23 - The average crossover path winding number for each winding number range.

6.1.2.3 A Sample of Crossover Paths

A review of the results of the crossover paths from experiments 4 and 5 discovered the paths with the least number of successful runs. The table in figure 6.24 lists the top five crossover paths, from experiment 4, with the least number of successful runs. For experiment 5, 1,446 paths have the least number of successful runs of 0. Therefore, the table in figure 6.25 lists the same top five crossover paths from experiment

4 but with the path's run results from experiment 5. In each table, each path entry lists its path number, path length, the path's number of successful runs, and the path's winding count. Recall from subsection 6.1.2.1, the smallest crossover path length is 13 and the largest crossover path length is 2644. Recall from subsection 6.1.2.2, the smallest winding number is 0 and the largest winding number is 65. All of the crossover paths in figures 6.24 and 6.25 have a path length greater than 2200. All of the crossover paths in figures 6.24 and 6.25 have a winding number greater than or equal to 57, within the upper end of the winding number range for the crossover paths.

Path Number	Path Length	Number of Successful Runs	Winding Number
9573	2630	14	61
5863	2424	17	61
9796	2591	18	57
5709	2589	19	57
814	2209	20	65

Figure 6.24 - Experiment 4's top five paths having the least number of successful runs.

Path Number	Path Length	Number of Successful Runs	Winding Number
9573	2630	0	61
5863	2424	0	61
9796	2591	0	57
5709	2589	3	57
814	2209	0	65

Figure 6.25 – The paths of figure 6.24 with the "number of successful runs" results from experiment 5.

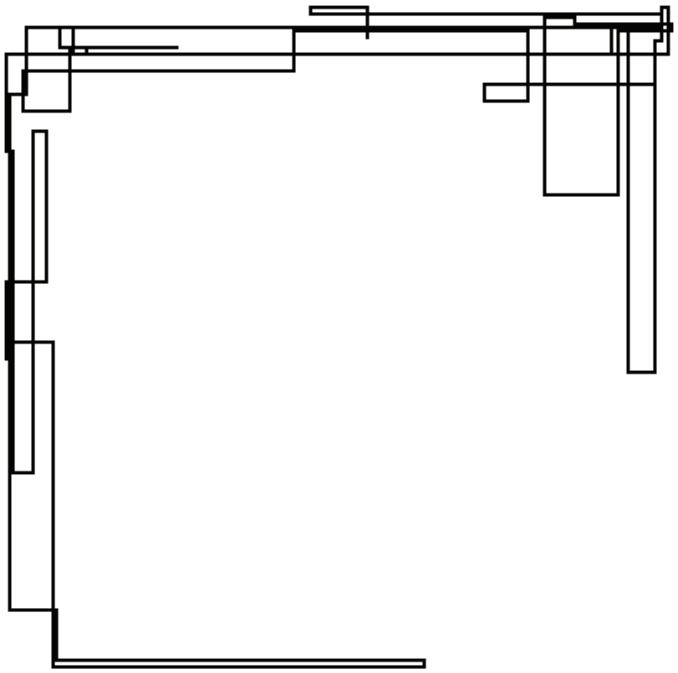


Figure 6.26 - Path 814, length: 2209, number of successful runs: 20, winding number: 65.

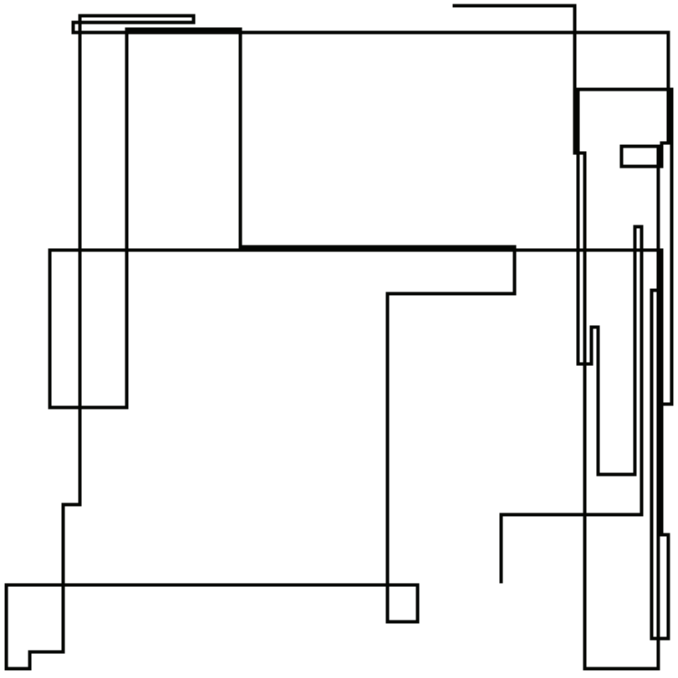


Figure 6.27 - Path 5709, length: 2589, number of successful runs: 19, winding number: 57.

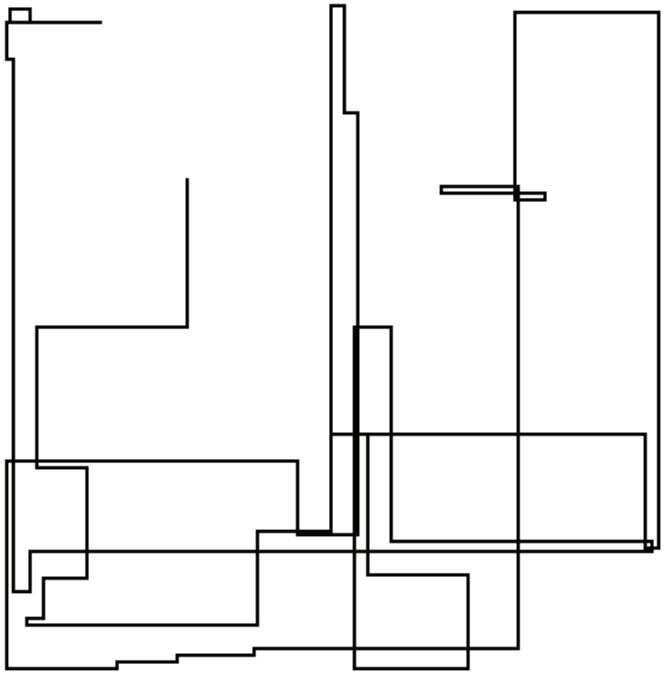


Figure 6.28 - Path 5863, length: 2424, number of successful runs: 17, winding number: 61.

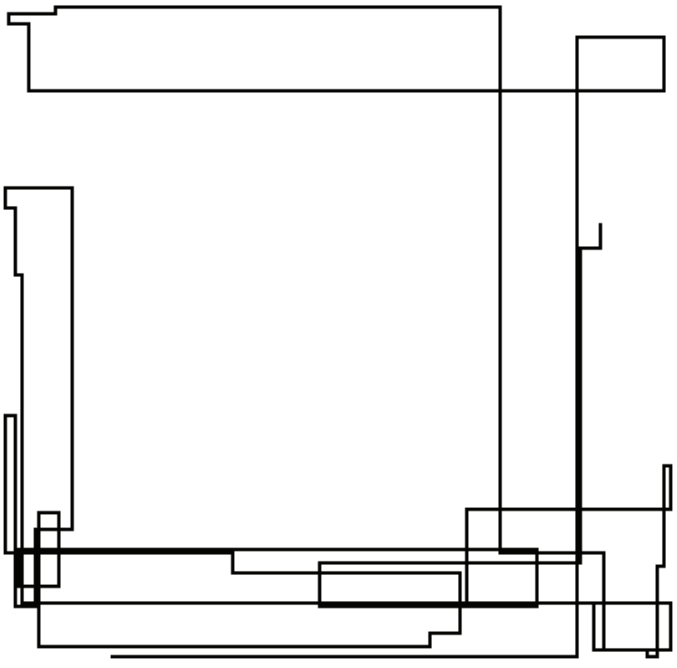


Figure 6.29 - Path 9573, length: 2630, number of successful runs: 14, winding number: 61.

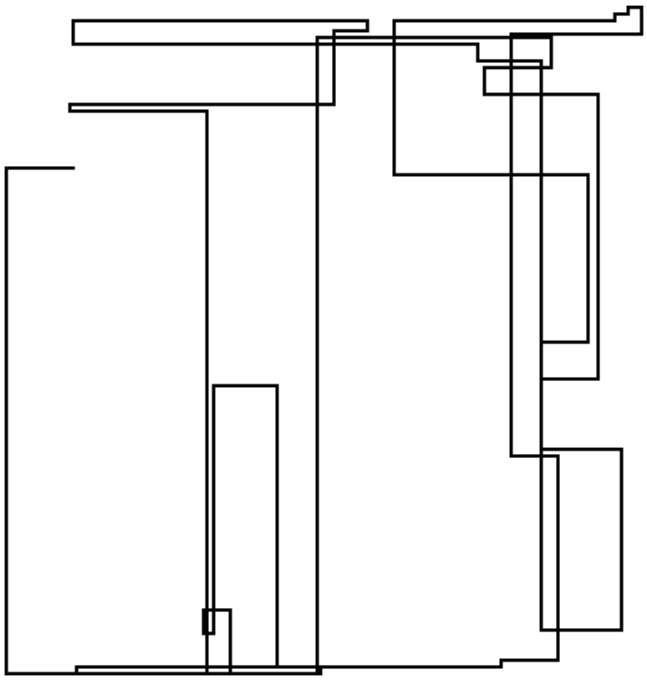


Figure 6.30 - Path 9796, length: 2591, number of successful runs: 18, winding number: 57.

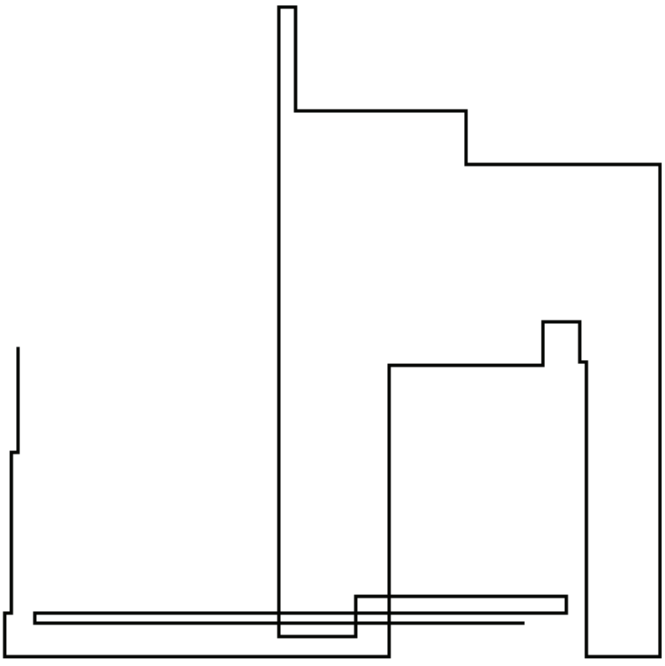


Figure 6.31 - Path 1910, length: 1400, number of successful runs: 100, winding number: 27.

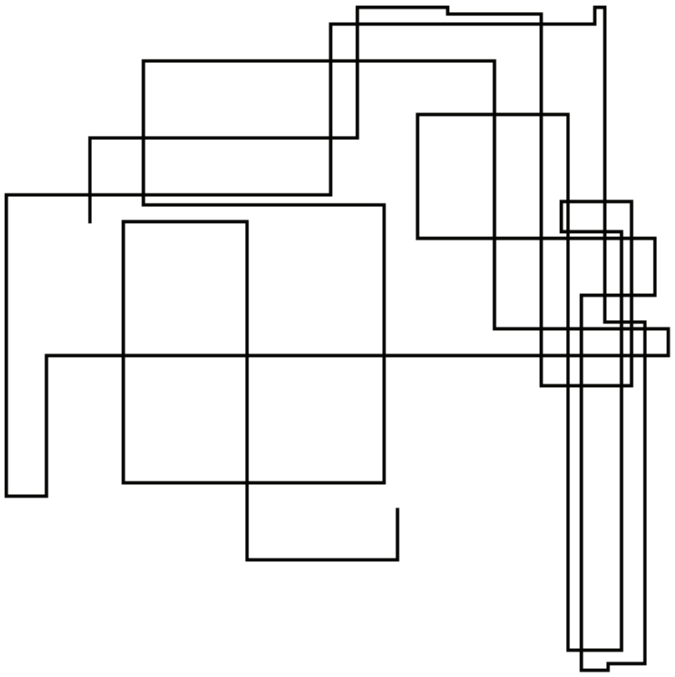


Figure 6.32 - Path 4273, length: 2639, number of successful runs: 100, winding number: 48.

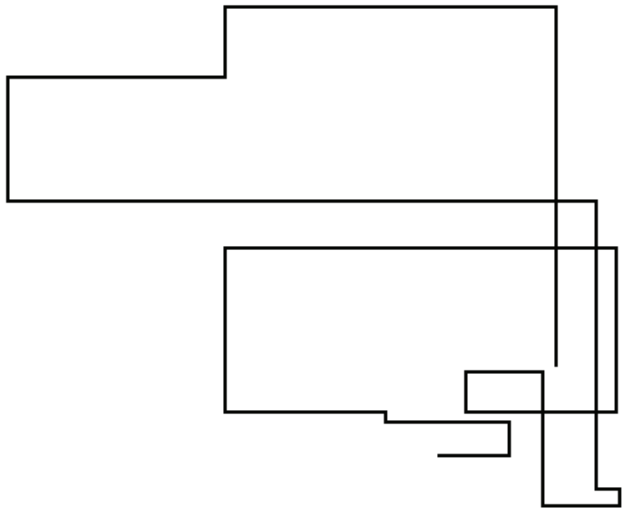


Figure 6.33 - Path 4369, length: 1081, number of successful runs: 100, winding number: 21.

Figures 6.26 through 6.33 display an image of each of the crossover paths in figures 6.24 and 6.25. Figure 6.26 is an image of crossover path 814. Figure 6.27 is an image of crossover path 5709. Figure 6.28 is an image of crossover path 5863. Figure 6.29 is an image of crossover path 9573. Figure 6.30 is an image of crossover path 9796. In contrast to these five paths, figures 6.31 through 6.33 display three crossover paths with the maximum possible number of successful runs, 100. Figure 6.31 is an image of crossover path 1910 with path length 1400 and winding number 27. Figure 6.32 is an image of crossover path 4273 with path length 2639 and winding number 48. Figure 6.33 is an image of crossover path 4369 with path length 1081 and winding number 21. The path lengths of two of the three successful paths, 1910 and 4273, are shorter than the path length of the unsuccessful paths in figures 6.24 and 6.25. But, the path length of the successful path 4369 is in the same range as the path length of the unsuccessful paths in figures 6.24 and 6.25. The small winding number of each of the three successful paths defines a simple geometry for these three paths making them easy for the agents to follow and consume. On the other hand, the large winding number of each of the five unsuccessful paths defines a complex geometry for these five paths making them difficult for the agents to follow and consume. The winding number is the true measure of the path's difficulty. The next subsection examines the results of the first signature recognition program experiment.

6.1.3 The First Signature Recognition Program Experiment

The examination of the first signature recognition program experiment begins in the next subsection with an analysis of the number of successful runs for the non-

crossover paths. Subsection 6.1.3.2 analyzes the number of unsuccessful runs for the non-crossover paths. Subsection 6.1.3.3 studies the average final generation numbers for the non-crossover paths. Finally, subsection 6.1.3.4 surveys the agents found by the signature recognition program during the first experiment to determine if the agents are contiguous agents.

6.1.3.1 Number of Successful Runs

For this experiment, the signature recognition program completes 100 runs of the genetic algorithm for each path. After processing all 10,000 paths, a separate data processing program collects data about each path, from the output files, computing the following four data values for each path: the number of successful runs of the path, the average number of agents per run, the average number of reachable states per agent, and the average generation number. In this experiment, the average number of successful runs for the paths is 99.64 and the standard deviation is 2.45. The formula, (number of successful runs – 99.64) / 2.45, computes the standard statistical Z score for each path's number of successful runs. The smallest Z score is -24.38 and the largest Z score is 0.15. There are 6 Z score ranges which are: $Z < -3$, $-3 \leq Z < -2$, $-2 \leq Z < -1$, $-1 \leq Z < 0$, $Z = 0$, $0 < Z \leq 1$. Count all of the paths whose Z score falls within each range. The chart in figure 6.34 illustrates the Z score counts for the non-crossover paths with the Z score ranges along the x-axis and the path count along the y-axis. The number of successful runs for a path is an integer. Because the average number of successful runs per path is 99.64, a Z score greater than 0 is a path whose number of successful runs is 100. From the chart in figure 6.34, 92.98% of the paths have 100 successful runs. Still, a path with $-3 \leq Z < 0$ is

a path with a number of successful runs between 93 and 99. Path number 1611 is the path with the least number of successful runs of 40 having a Z score of -24.38. The next subsection analyzes the number of unsuccessful runs for the non-crossover paths.

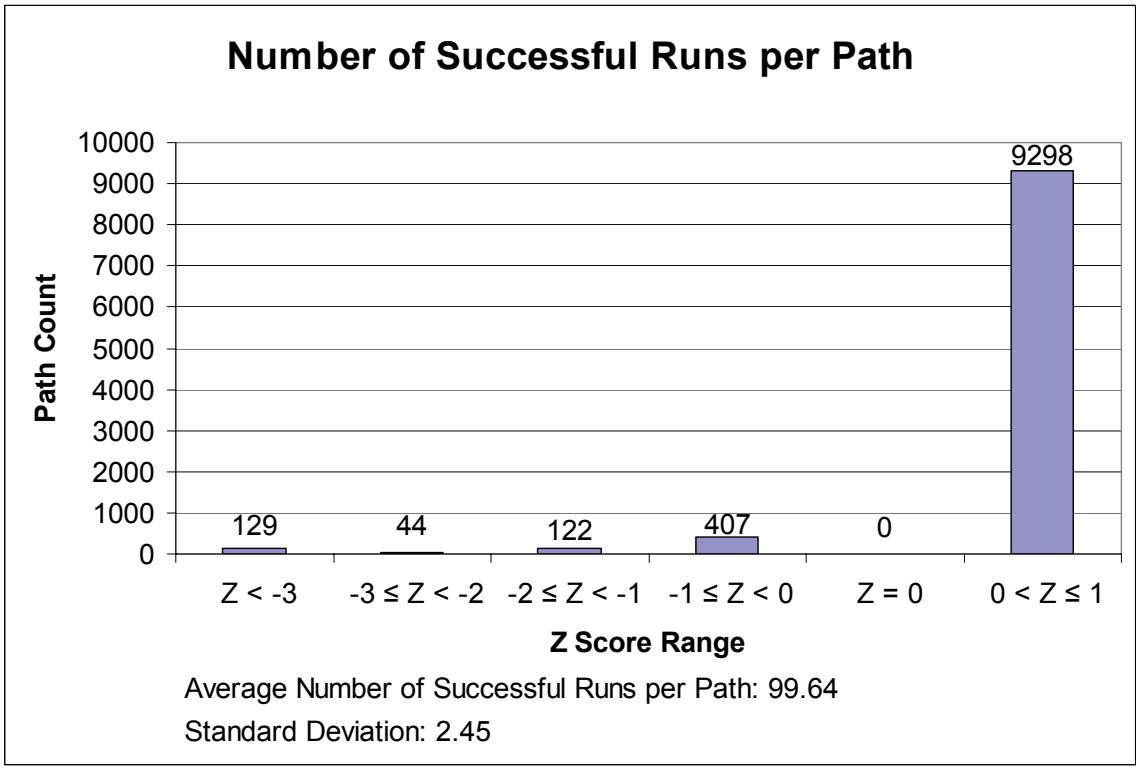


Figure 6.34 - Experiment 1's path counts for the number of successful runs per path.

6.1.3.2 Number of Unsuccessful Runs

After the first experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the number of unsuccessful runs for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length. The length of

runCounts is 2645 with its indices ranging from 0 to 2644. The i^{th} element of runCounts corresponds to path length i. The program initializes each element in runCounts to zero. For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length.

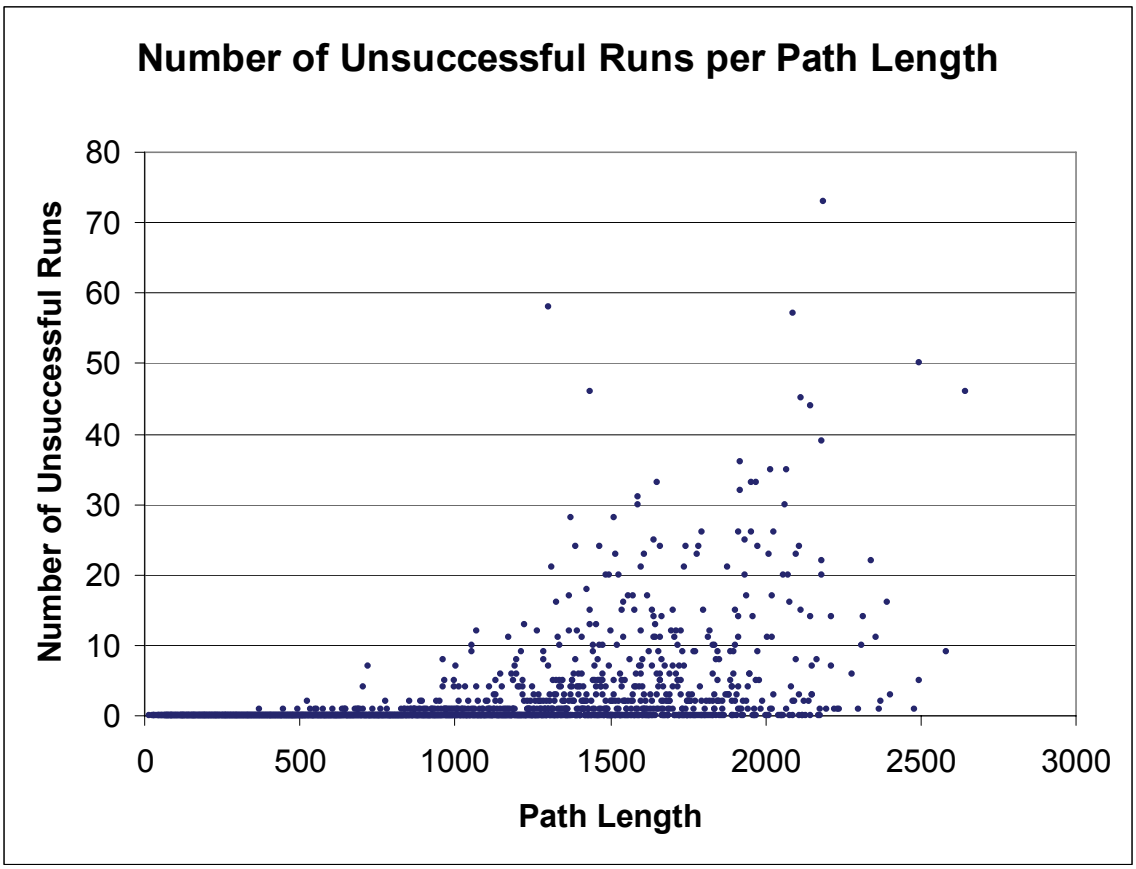


Figure 6.35 - The number of unsuccessful runs per path length for experiment 1.

The scatter chart in figure 6.35 illustrates the number of unsuccessful runs for the non-crossover paths with the number of unsuccessful runs along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the number of unsuccessful runs for a particular path length. The line at 0 unsuccessful runs is actually

a series of points for a large number of path lengths. As expected, the longer path lengths have more unsuccessful runs. But, there are some path lengths throughout the path length range that have unsuccessful runs. The path's geometry also plays a part in a path's number of unsuccessful runs. The next subsection analyzes the average final generation number for the non-crossover paths.

6.1.3.3 Average Final Generation Number

After the first experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the average final generation number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of successful runs for each unique path length as well as another one dimensional array, called generationSums, to store the summation of the final generation numbers for the successful runs of each unique path length. The length of each array, runCounts and generationSums, is 2645 with their indices ranging from 0 to 2644. The i^{th} element of each array, runCounts and generationSums, corresponds to path length i. The program initializes each element in runCounts and generationSums to zero.

For each path, the program retrieves the path's length and computes the number of successful runs for that path, adding the number of successful runs to the element in runCounts corresponding to the path's length. For each successful run of a path, the program retrieves the path's length and the final generation number of the successful run, adding the final generation number to the element in generationSums corresponding to the path's length. After the data processing program processes all 10,000 non-crossover

paths, the program computes the average final generation number of each path length i as generationSums[i] / runCounts[i], where runCounts[i] > 0.

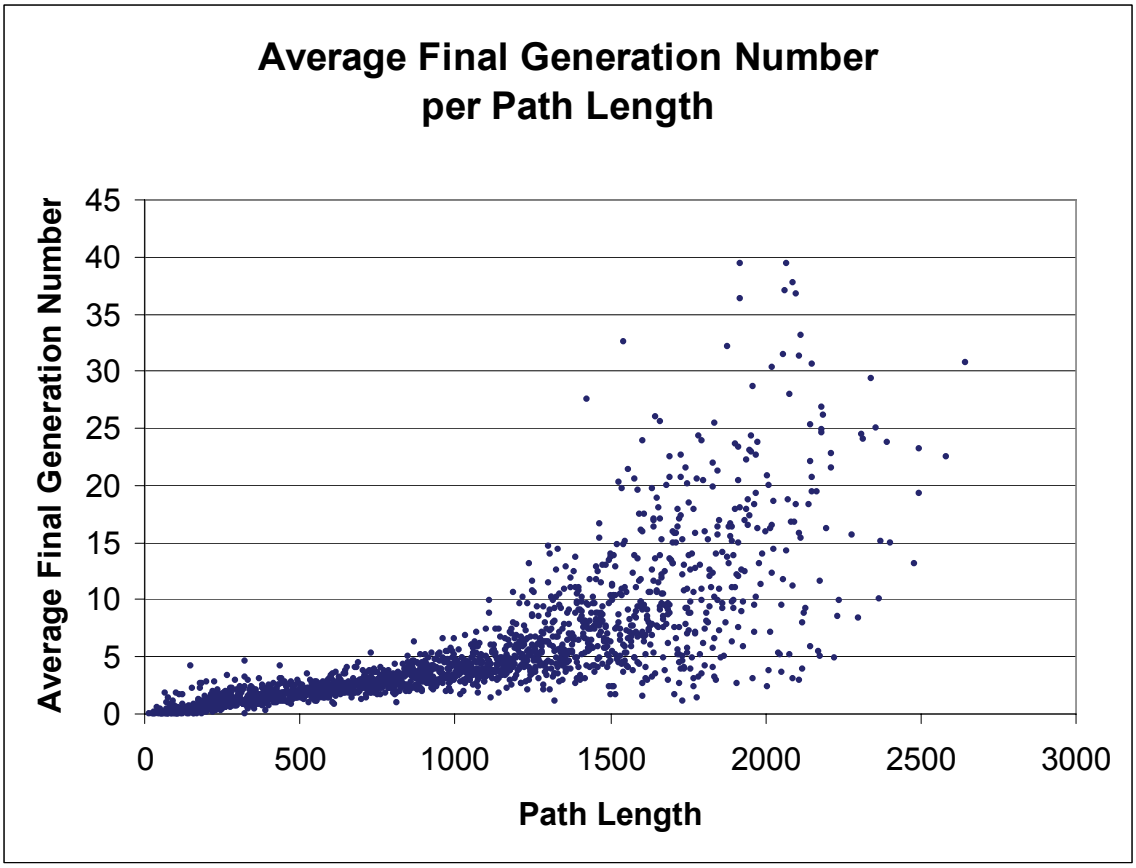


Figure 6.36 - The average final generation number per path length for experiment 1.

The scatter chart in figure 6.36 illustrates the average final generation number for the non-crossover paths with the average final generation number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average final generation number for a particular path length. As expected, the final generation number increases in tandem with the path length. The final generation numbers fall within a narrow range for the path length range of 13 to approximately 1200 meaning the path geometry for neighboring path lengths, within this range, remains consistent such that the

signature recognition genetic algorithm requires the same amount of work to find agents to recognize those paths. For paths whose length is above 1200, the final generation numbers fall within a larger range meaning the path geometry for neighboring path lengths varies greatly such that the signature recognition genetic algorithm requires varying degrees of work to find agents to recognize those paths. The next subsection examines the agents to determine if any of them are contiguous agents.

6.1.3.4 Contiguous Agents

Determine if an agent written to the first experiment's output file consumes its path in full contiguous order from beginning to end. A full contiguous order means the agent consumes each path square in order from the beginning path square to the end path square. An agent can also consume the path in a partial contiguous order. A partial contiguous order means the agent consumes segments of the path in a full contiguous order but the agent consumes the path squares between segments out of order. The agent consumes path squares within a segment in a full contiguous order.

Count all of the agents written to the output file for a path. The fitness function computeFSMPCFitness, section 2.3 in chapter 2, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. The function computeFSMPCFitness can determine if an agent consumes a path in full contiguous order because full contiguous order is a special case of partial contiguous order. The fitness value returned by computeFSMPCFitness is the number of path squares consumed in a partial contiguous order. Therefore, if an agent's fitness value computed by computeFSMPCFitness is equal to the path's length then the agent

consumed the path in full contiguous order; hence, the agent is a contiguous agent. Count all of the path's contiguous agents. Divide the path's number of contiguous agents by the path's total number of agents, multiplied by 100 to compute the path's percentage of contiguous agents.

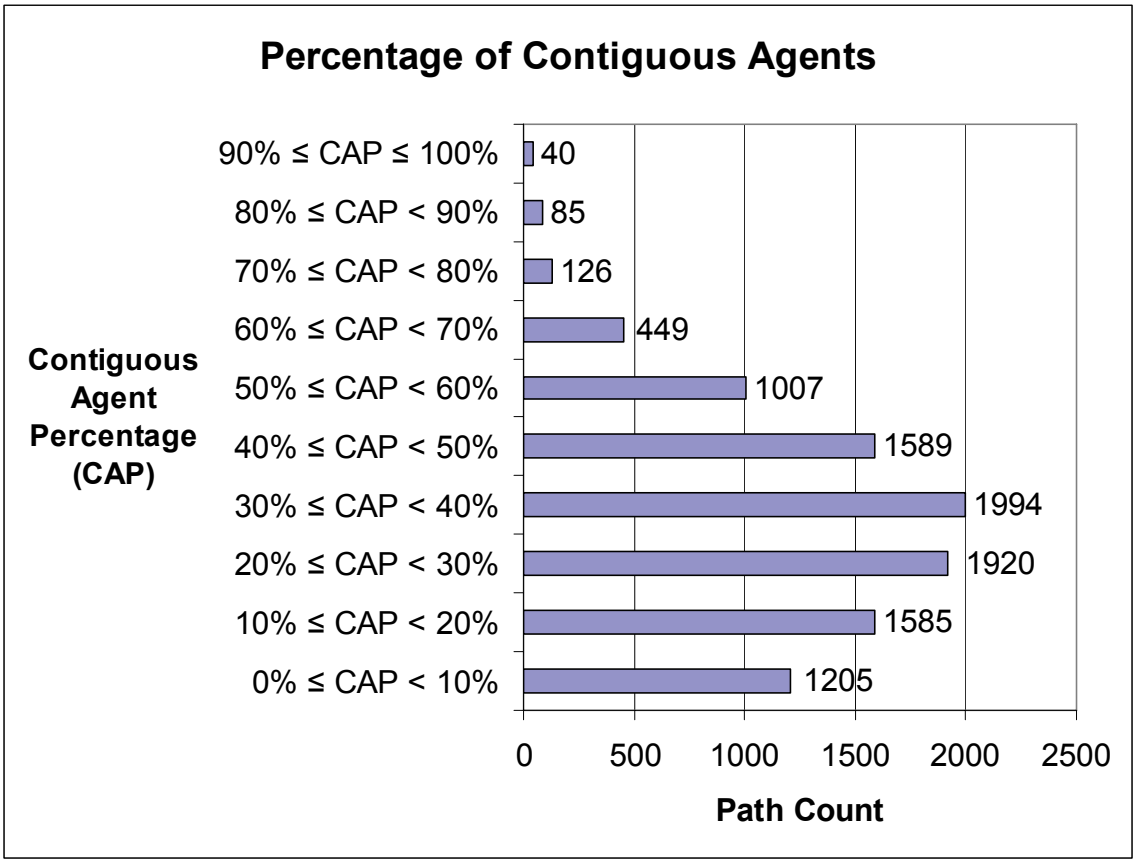


Figure 6.37 - Experiment 1's path counts for percentage of contiguous agents.

The contiguous agent percentage (CAP) runs from 0% to 100% divided into 10 ranges which are: 0% ≤ CAP < 10%, 10% ≤ CAP < 20%, 20% ≤ CAP < 30%, 30% ≤ CAP < 40%, 40% ≤ CAP < 50%, 50% ≤ CAP < 60%, 60% ≤ CAP < 70%, 70% ≤ CAP < 80%, 80% ≤ CAP < 90%, 90% ≤ CAP ≤ 100%. Count all of the paths whose contiguous agent percentage falls within each range. The chart in figure 6.37 illustrates the path

count for the contiguous agent percentage with the contiguous agent percentage ranges along the y-axis and the path count along the x-axis. The chart of the path counts form a bell curve with the top of the curve at the 30% to 40% range indicating a well distributed set of paths. An examination of the percentage values reveals that 63 of the 10,000 paths had 0 contiguous agents. The next subsection examines the results of the second signature recognition program experiment.

6.1.4 The Second Signature Recognition Program Experiment

The examination of the second signature recognition program experiment begins in the next subsection with an analysis of the number of successful runs for the non-crossover paths. Subsection 6.1.4.2 analyzes the number of unsuccessful runs for the non-crossover paths. Subsection 6.1.4.3 studies the average final generation numbers for the non-crossover paths. Finally, subsection 6.1.4.4 surveys the agents found by the signature recognition program during the second experiment to determine if the agents are contiguous agents.

6.1.4.1 Number of Successful Runs

Recall from subsection 2.6.3 in chapter 2, for this experiment, each agent completes two part processing before computing the agent's fitness value. The first part of processing an agent determines the states in the agent that are reachable from the start state 0. Compaction is the second part of processing the agent. Compaction groups all of the reachable states together at the top of the agent's finite state table and groups all of the non-reachable states at the bottom of the agent's table.

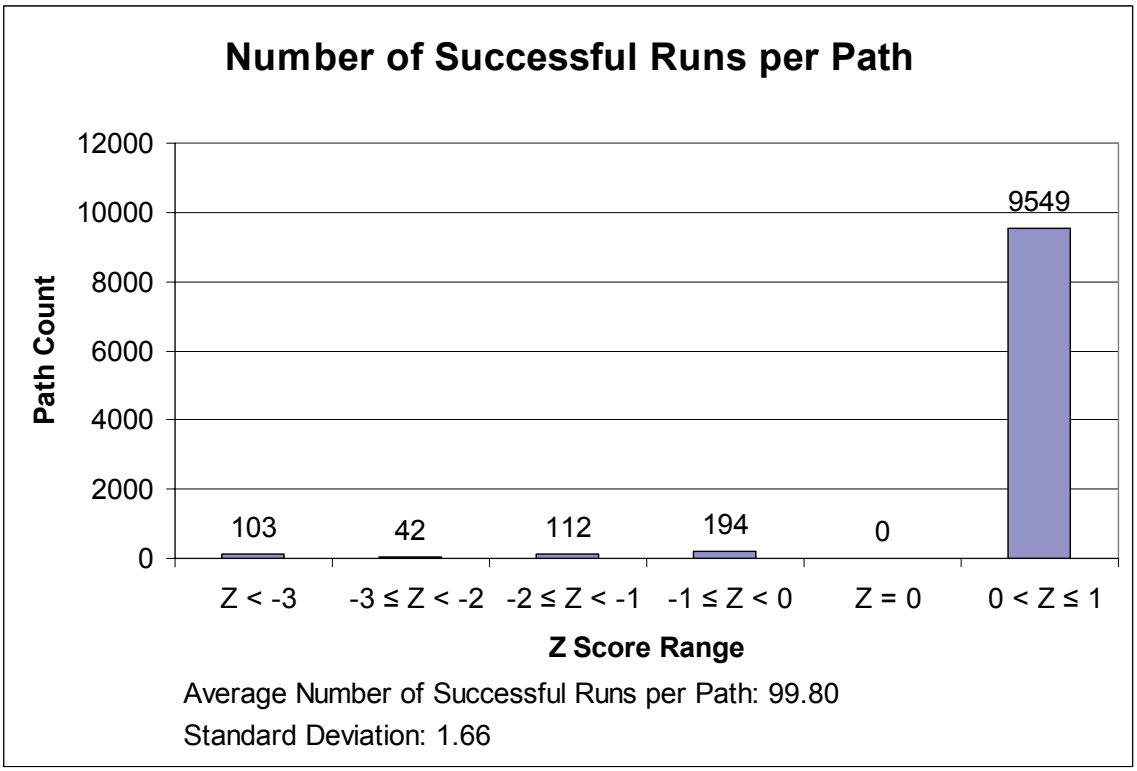


Figure 6.38 - Experiment 2's path counts for the number of successful runs per path.

For this experiment, the signature recognition program completes 100 runs of the genetic algorithm for each path. After processing all 10,000 paths, a separate data processing program collects data about each path, from the output files, computing the following four data values for each path: the number of successful runs of the path, the average number of agents per run, the average number of reachable states per agent, and the average generation number. In this experiment, the average number of successful runs for the paths is 99.80 and the standard deviation is 1.66. The formula, (number of successful runs – 99.80) / 1.66, computes the standard statistical Z score for each path's number of successful runs. The smallest Z score is -36.56 and the largest Z score is 0.12. There are 6 Z score ranges which are: $Z < -3$, $-3 \leq Z < -2$, $-2 \leq Z < -1$, $-1 \leq Z < 0$, $Z = 0$, $0 < Z \leq 1$. Count all of the paths whose Z score falls within each range. The chart in figure

6.38 illustrates the Z score counts for the non-crossover paths with the Z score ranges along the x-axis and the path count along the y-axis. The number of successful runs for a path is an integer. Because the average number of successful runs per path is 99.80, a Z score greater than 0 is a path whose number of successful runs is 100. From the chart in figure 6.38, 95.49% of the paths have 100 successful runs. Still, a path with $-3 \leq Z < 0$ is a path with a number of successful runs between 95 and 99. Path number 1611 is the path with the least number of successful runs of 39 having a Z score of -36.56. The next subsection analyzes the number of unsuccessful runs for the non-crossover paths.

6.1.4.2 Number of Unsuccessful Runs

After the second experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the number of unsuccessful runs for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length. The length of runCounts is 2645 with its indices ranging from 0 to 2644. The i^{th} element of runCounts corresponds to path length i. The program initializes each element in runCounts to zero. For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length.

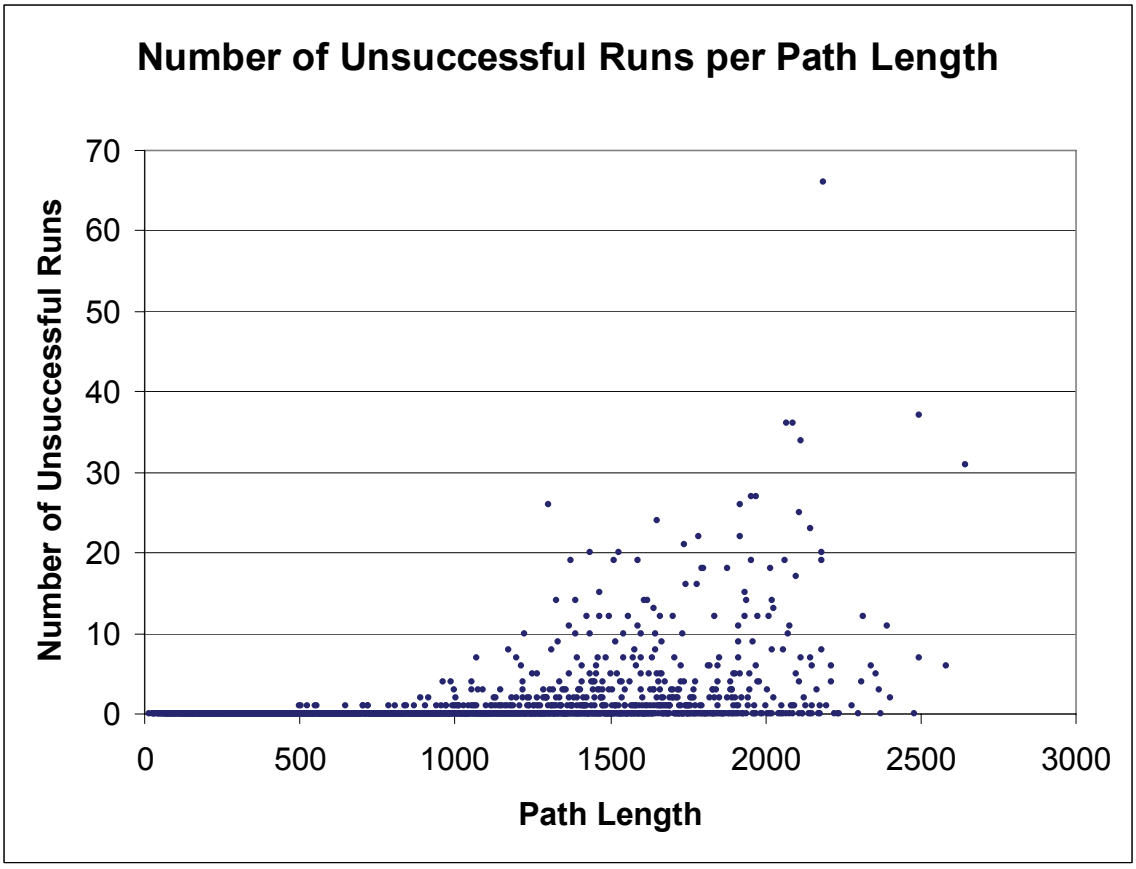


Figure 6.39 - The number of unsuccessful runs per path length for experiment 2.

The scatter chart in figure 6.39 illustrates the number of unsuccessful runs for the non-crossover paths with the number of unsuccessful runs along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the number of unsuccessful runs for a particular path length. The line at 0 unsuccessful runs is actually a series of points for a large number of path lengths. As expected, the longer path lengths have more unsuccessful runs. But, there are some path lengths throughout the path length range that have unsuccessful runs. The path's geometry also plays a part in a path's number of unsuccessful runs. A comparison of the chart of unsuccessful runs for the second experiment to the chart of unsuccessful runs for the first experiment reveals the second experiment having fewer unsuccessful runs than the first experiment. The

compaction performed on the agents of the second experiment improves the performance of the signature recognition genetic algorithm. The next subsection analyzes the average final generation number for the non-crossover paths.

6.1.4.3 Average Final Generation Number

After the second experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the average final generation number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of successful runs for each unique path length as well as another one dimensional array, called generationSums, to store the summation of the final generation numbers for the successful runs of each unique path length. The length of each array, runCounts and generationSums, is 2645 with their indices ranging from 0 to 2644. The i^{th} element of each array, runCounts and generationSums, corresponds to path length i. The program initializes each element in runCounts and generationSums to zero.

For each path, the program retrieves the path's length and computes the number of successful runs for that path, adding the number of successful runs to the element in runCounts corresponding to the path's length. For each successful run of a path, the program retrieves the path's length and the final generation number of the successful run, adding the final generation number to the element in generationSums corresponding to the path's length. After the data processing program processes all 10,000 non-crossover paths, the program computes the average final generation number of each path length i as generationSums[i] / runCounts[i], where runCounts[i] > 0.

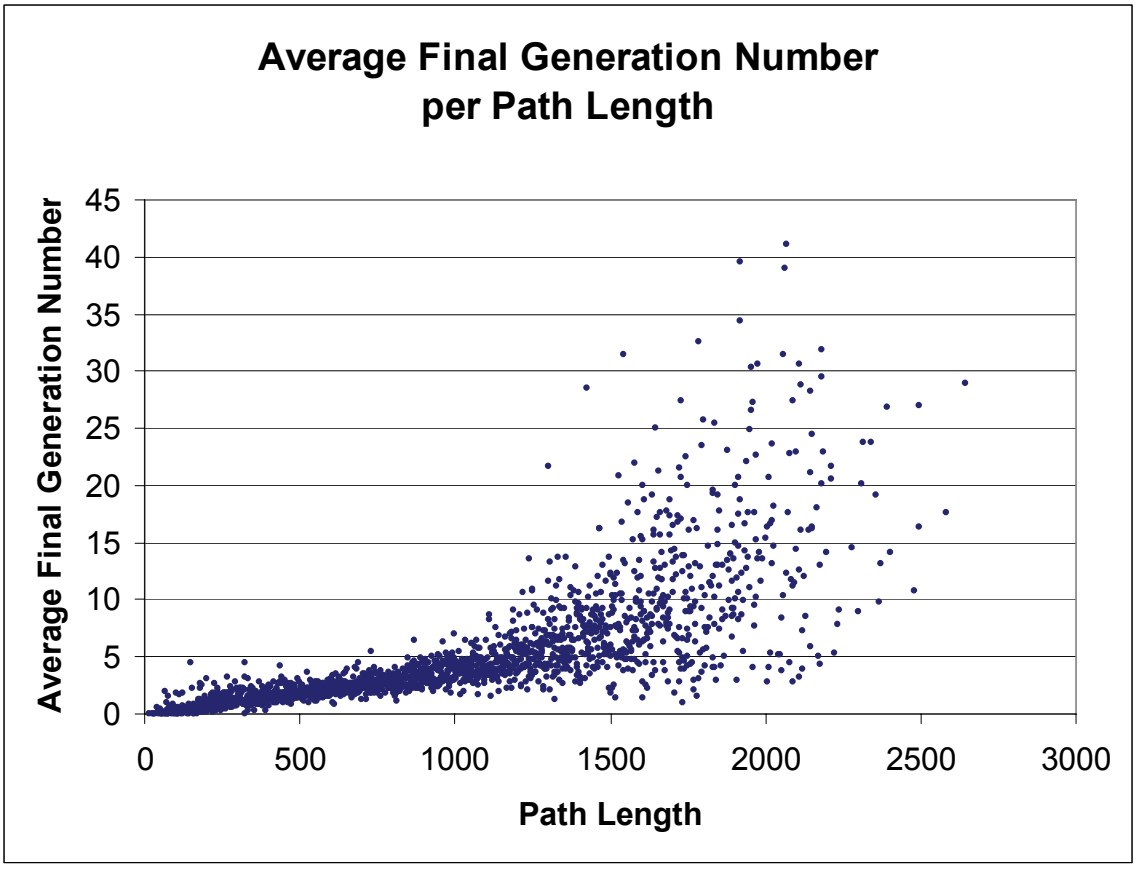


Figure 6.40 - The average final generation number per path length for experiment 2.

The scatter chart in figure 6.40 illustrates the average final generation number for the non-crossover paths with the average final generation number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average final generation number for a particular path length. As expected, the final generation number increases in tandem with the path length. The final generation numbers fall within a narrow range for the path length range of 13 to approximately 1200 meaning the path geometry for neighboring path lengths, within this range, remains consistent such that the signature recognition genetic algorithm requires the same amount of work to find agents to recognize those paths. For paths whose length is above 1200, the final generation numbers fall within a larger range meaning the path geometry for neighboring path

lengths varies greatly such that the signature recognition genetic algorithm requires varying degrees of work to find agents to recognize those paths. The next subsection examines the agents to determine if any of them are contiguous agents.

6.1.4.4 Contiguous Agents

Determine if an agent written to the second experiment's output file consumes its path in full contiguous order from beginning to end. A full contiguous order means the agent consumes each path square in order from the beginning path square to the end path square. An agent can also consume the path in a partial contiguous order. A partial contiguous order means the agent consumes segments of the path in a full contiguous order but the agent consumes the path squares between segments out of order. The agent consumes path squares within a segment in a full contiguous order.

Count all of the agents written to the output file for a path. The fitness function computeFSMPCFitness, section 2.3 in chapter 2, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. The function computeFSMPCFitness can determine if an agent consumes a path in full contiguous order because full contiguous order is a special case of partial contiguous order. The fitness value returned by computeFSMPCFitness is the number of path squares consumed in partial contiguous order. Therefore, if an agent's fitness value computed by computeFSMPCFitness is equal to the path's length then the agent consumed the path in full contiguous order; hence, the agent is a contiguous agent. Count all of the path's contiguous agents. Divide the path's number of contiguous agents

by the path's total number of agents, multiplied by 100 to compute the path's percentage of contiguous agents.

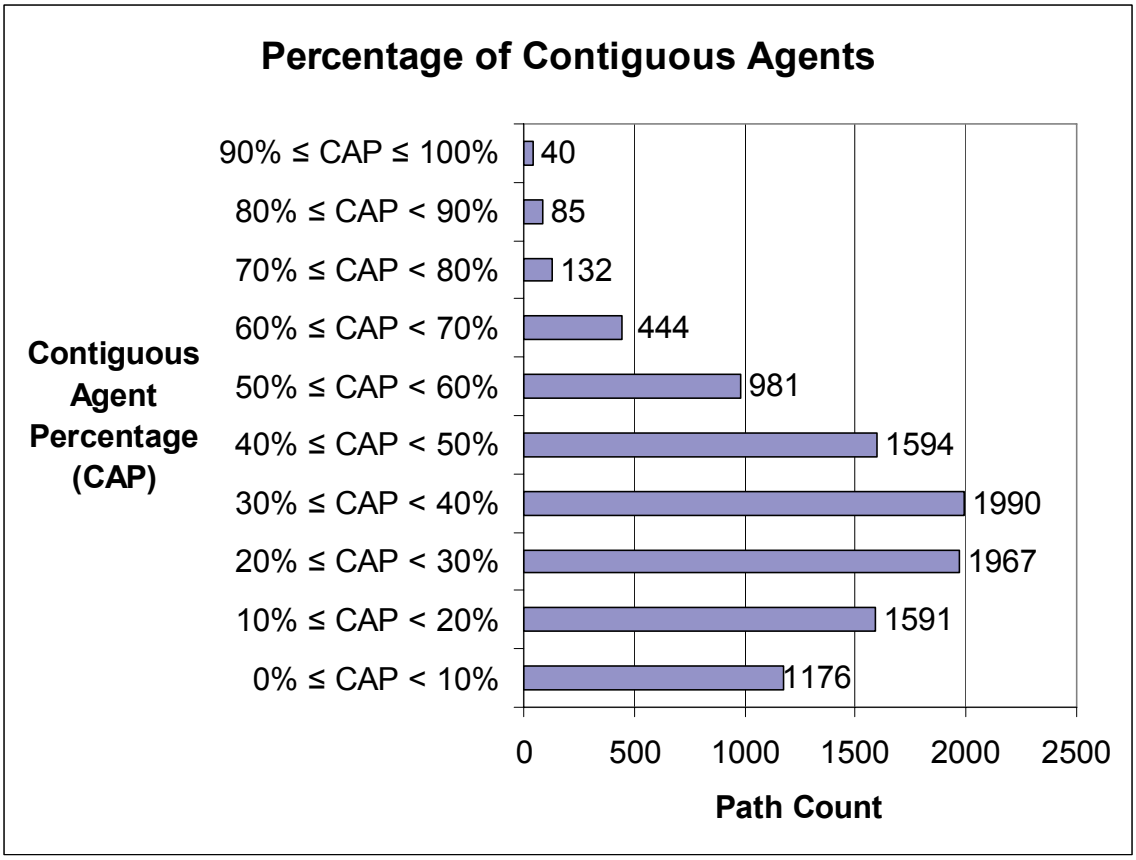


Figure 6.41 - Experiment 2's path counts for percentage of contiguous agents.

The contiguous agent percentage (CAP) runs from 0% to 100% divided into 10 ranges which are: 0% ≤ CAP < 10%, 10% ≤ CAP < 20%, 20% ≤ CAP < 30%, 30% ≤ CAP < 40%, 40% ≤ CAP < 50%, 50% ≤ CAP < 60%, 60% ≤ CAP < 70%, 70% ≤ CAP < 80%, 80% ≤ CAP < 90%, 90% ≤ CAP ≤ 100%. Count all of the paths whose contiguous agent percentage falls within each range. The chart in figure 6.41 illustrates the path count for the contiguous agent percentage with the contiguous agent percentage ranges along the y-axis and the path count along the x-axis. The chart of the path counts form a

bell curve with the top of the curve at the 30% to 40% range indicating a well distributed set of paths. An examination of the percentage values reveals that 60 of the 10,000 paths had 0 contiguous agents. The next subsection examines the results of the third signature recognition program experiment.

6.1.5 The Third Signature Recognition Program Experiment

The examination of the third signature recognition program experiment begins in the next subsection with an analysis of the number of successful runs for the non-crossover paths. Subsection 6.1.5.2 analyzes the number of unsuccessful runs for the non-crossover paths. Subsection 6.1.5.3 studies the average final generation numbers for the non-crossover paths. Finally, subsection 6.1.5.4 surveys the agents found by the signature recognition program during the third experiment to determine if the agents are contiguous agents.

6.1.5.1 Number of Successful Runs

Recall from subsection 2.6.3 in chapter 2, for this experiment, each agent completes two part processing before computing the agent's fitness value. The first part of processing an agent determines the states in the agent that are reachable from the start state 0. Spreading is the second part of processing the agent. Spreading evenly spaces all of the reachable states and non-reachable states throughout the agent's finite state table.

For this experiment, the signature recognition program completes 100 runs of the genetic algorithm for each path. After processing all 10,000 paths, a separate data processing program collects data about each path, from the output files, computing the

following four data values for each path: the number of successful runs of the path, the average number of agents per run, the average number of reachable states per agent, and the average generation number. In this experiment, the average number of successful runs for the paths is 99.75 and the standard deviation is 1.99. The formula, (number of successful runs - 99.75) / 1.99, computes the standard statistical Z score for each path's number of successful runs. The smallest Z score is -25.52 and the largest Z score is 0.13. There are 6 Z score ranges which are: $Z < -3$, $-3 \leq Z < -2$, $-2 \leq Z < -1$, $-1 \leq Z < 0$, $Z = 0$, $0 < Z \leq 1$. Count all of the paths whose Z score falls within each range.

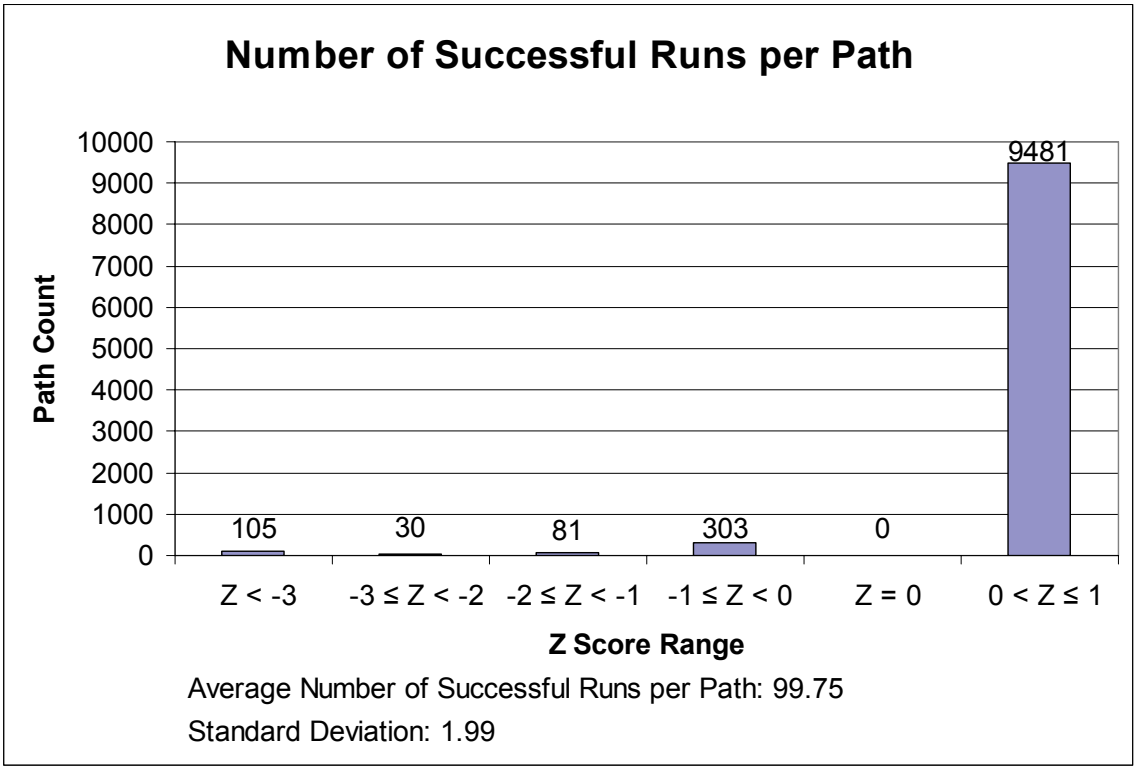


Figure 6.42 - Experiment 3's path counts for the number of successful runs per path.

The chart in figure 6.42 illustrates the Z score counts for the non-crossover paths with the Z score ranges along the x-axis and the path count along the y-axis. The number

of successful runs for a path is an integer. Because the average number of successful runs per path is 99.75, a Z score greater than 0 is a path whose number of successful runs is 100. From the chart in figure 6.42, 94.81% of the paths have 100 successful runs. Still, a path with $-3 \leq Z < 0$ is a path with a number of successful runs between 94 and 99. Path number 5809 is the path with the least number of successful runs of 49 having a Z score of -25.52. The next subsection analyzes the number of unsuccessful runs for the non-crossover paths.

6.1.5.2 Number of Unsuccessful Runs

After the third experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the number of unsuccessful runs for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length. The length of runCounts is 2645 with its indices ranging from 0 to 2644. The i^{th} element of runCounts corresponds to path length i. The program initializes each element in runCounts to zero. For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length.

The scatter chart in figure 6.43 illustrates the number of unsuccessful runs for the non-crossover paths with the number of unsuccessful runs along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the number of unsuccessful runs for a particular path length. The line at 0 unsuccessful runs is actually

a series of points for a large number of path lengths. As expected, the longer path lengths have more unsuccessful runs. But, there are some path lengths throughout the path length range that have unsuccessful runs. The path's geometry also plays a part in a path's number of unsuccessful runs. Just like the second experiment, a comparison of the chart of unsuccessful runs for the third experiment to the chart of unsuccessful runs for the first experiment reveals the third experiment having fewer unsuccessful runs than the first experiment. The spreading performed on the agents of the third experiment improves the performance of the signature recognition genetic algorithm. The next subsection analyzes the average final generation number for the non-crossover paths.

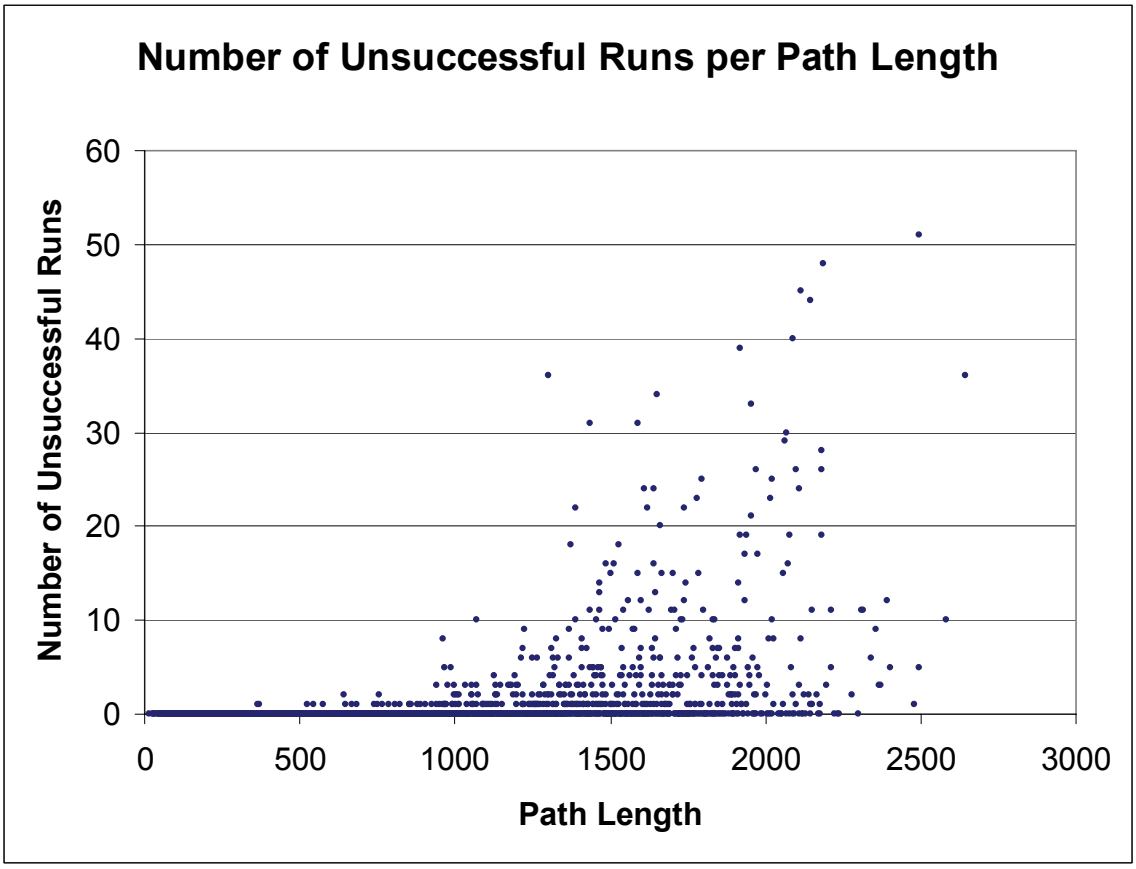


Figure 6.43 - The number of unsuccessful runs per path length for experiment 3.

6.1.5.3 Average Final Generation Number

After the third experiment processes all 10,000 non-crossover paths, a separate data processing program collects data about the average final generation number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of successful runs for each unique path length as well as another one dimensional array, called generationSums, to store the summation of the final generation numbers for the successful runs of each unique path length. The length of each array, runCounts and generationSums, is 2645 with their indices ranging from 0 to 2644. The i^{th} element of each array, runCounts and generationSums, corresponds to path length i. The program initializes each element in runCounts and generationSums to zero.

For each path, the program retrieves the path's length and computes the number of successful runs for that path, adding the number of successful runs to the element in runCounts corresponding to the path's length. For each successful run of a path, the program retrieves the path's length and the final generation number of the successful run, adding the final generation number to the element in generationSums corresponding to the path's length. After the data processing program processes all 10,000 non-crossover paths, the program computes the average final generation number of each path length i as generationSums[i] / runCounts[i], where runCounts[i] > 0.

The scatter chart in figure 6.44 illustrates the average final generation number for the non-crossover paths with the average final generation number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average final

generation number for a particular path length. As expected, the final generation number increases in tandem with the path length. The final generation numbers fall within a narrow range for the path length range of 13 to approximately 1200 meaning the path geometry for neighboring path lengths, within this range, remains consistent such that the signature recognition genetic algorithm requires the same amount of work to find agents to recognize those paths. For paths whose length is above 1200, the final generation numbers fall within a larger range meaning the path geometry for neighboring path lengths varies greatly such that the signature recognition genetic algorithm requires varying degrees of work to find agents to recognize those paths. The next subsection examines the agents to determine if any of them are contiguous agents.

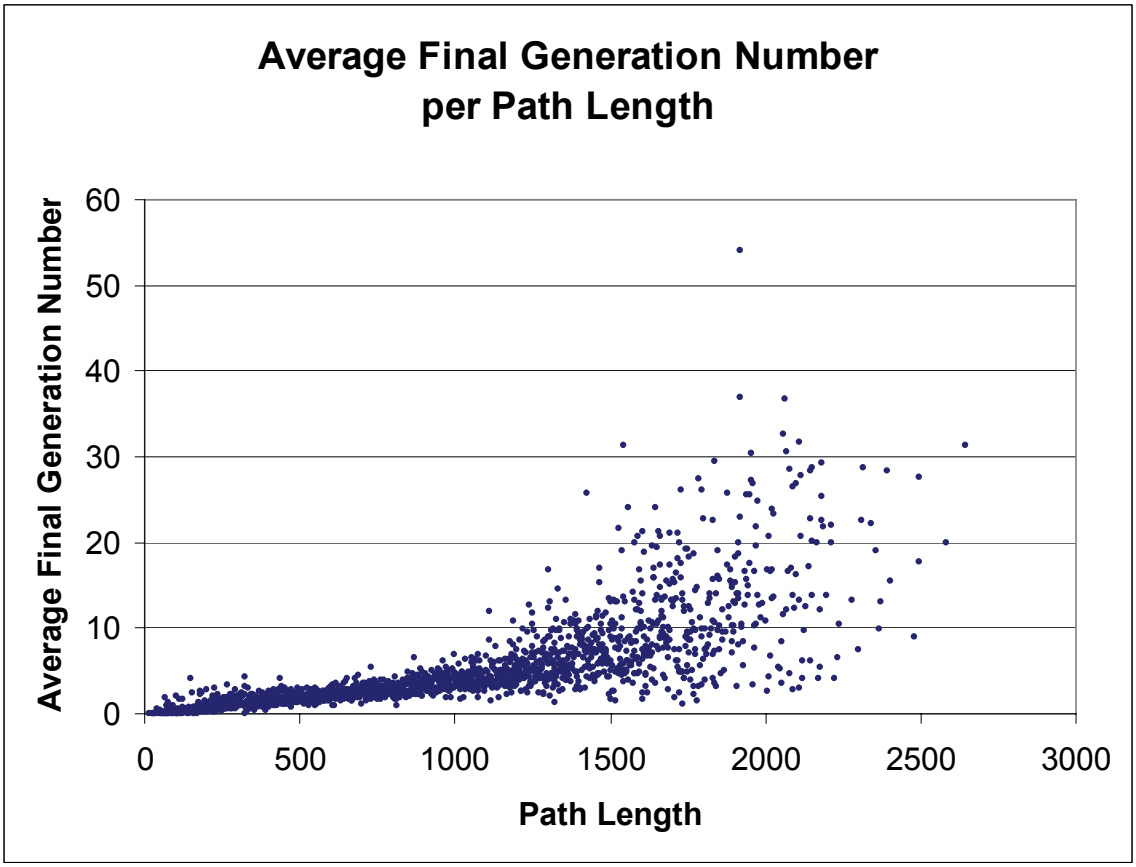


Figure 6.44 - The average final generation number per path length for experiment 3.

6.1.5.4 Contiguous Agents

Determine if an agent written to the third experiment's output file consumes its path in full contiguous order from beginning to end. A full contiguous order means the agent consumes each path square in order from the beginning path square to the end path square. An agent can also consume the path in a partial contiguous order. A partial contiguous order means the agent consumes segments of the path in a full contiguous order but the agent consumes the path squares between segments out of order. The agent consumes path squares within a segment in a full contiguous order.

Count all of the agents written to the output file for a path. The fitness function computeFSMPCFitness, section 2.3 in chapter 2, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. The function computeFSMPCFitness can determine if an agent consumes a path in full contiguous order because full contiguous order is a special case of partial contiguous order. The fitness value returned by computeFSMPCFitness is the number of path squares consumed in a partial contiguous order. Therefore, if an agent's fitness value computed by computeFSMPCFitness is equal to the path's length then the agent consumed the path in full contiguous order; hence, the agent is a contiguous agent. Count all of the path's contiguous agents. Divide the path's number of contiguous agents by the path's total number of agents, multiplied by 100 to compute the path's percentage of contiguous agents.

The contiguous agent percentage (CAP) runs from 0% to 100% divided into 10 ranges which are: $0\% \leq CAP < 10\%$, $10\% \leq CAP < 20\%$, $20\% \leq CAP < 30\%$, $30\% \leq$

CAP < 40%, 40% ≤ CAP < 50%, 50% ≤ CAP < 60%, 60% ≤ CAP < 70%, 70% ≤ CAP < 80%, 80% ≤ CAP < 90%, 90% ≤ CAP ≤ 100%. Count all of the paths whose contiguous agent percentage falls within each range. The chart in figure 6.45 illustrates the path count for the contiguous agent percentage with the contiguous agent percentage ranges along the y-axis and the path count along the x-axis. The chart of the path counts form a bell curve with the top of the curve at the 30% to 40% range indicating a well distributed set of paths. An examination of the percentage values reveals that 70 of the 10,000 paths had 0 contiguous agents. The next subsection examines the results of the fourth signature recognition program experiment.

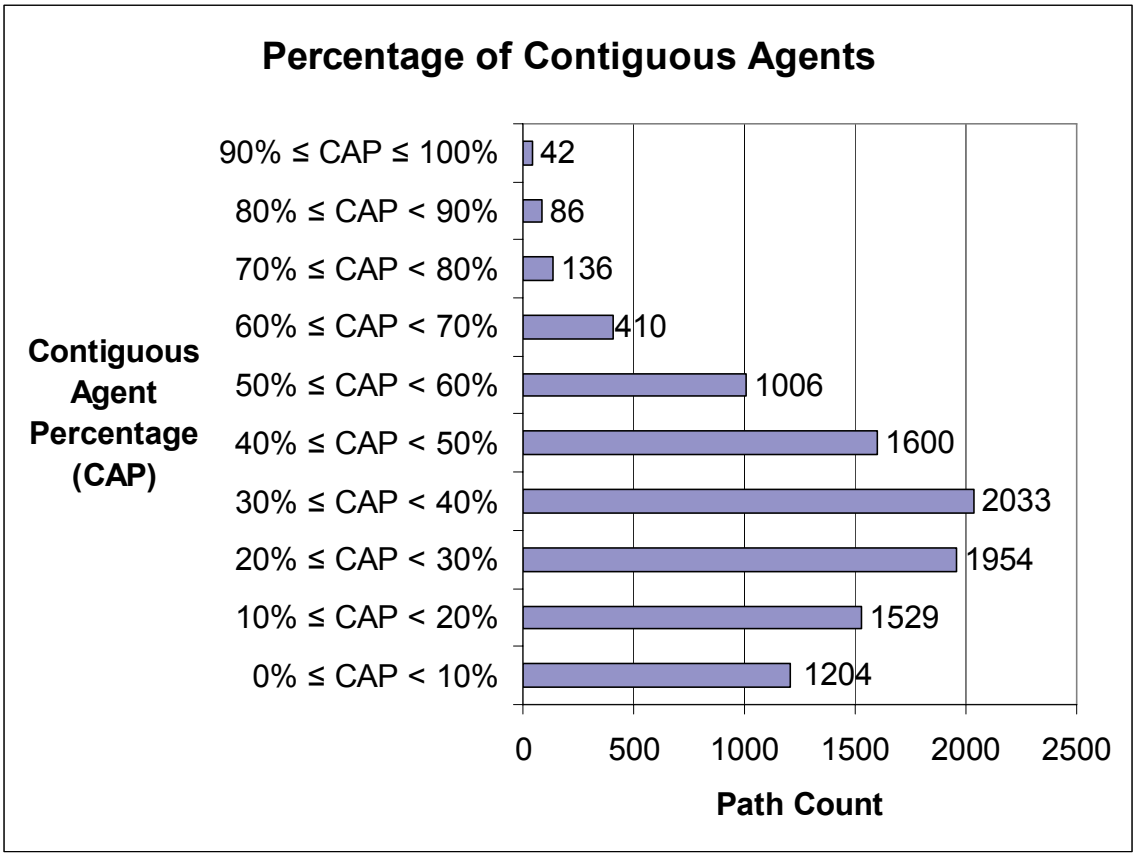


Figure 6.45 - Experiment 3's path counts for percentage of contiguous agents.

6.1.6 The Fourth Signature Recognition Program Experiment

The examination of the fourth signature recognition program experiment begins in the next subsection with an analysis of the number of successful runs for the crossover paths. Subsection 6.1.6.2 analyzes the number of unsuccessful runs for the crossover paths. Subsection 6.1.6.3 studies the average final generation numbers for the crossover paths. Finally, subsection 6.1.6.4 surveys the agents found by the signature recognition program during the fourth experiment to determine if the agents are contiguous agents.

6.1.6.1 Number of Successful Runs

For this experiment, the signature recognition program completes 100 runs of the genetic algorithm for each path. After processing all 10,000 paths, a separate data processing program collects data about each path, from the output files, computing the following four data values for each path: the number of successful runs of the path, the average number of agents per run, the average number of reachable states per agent, and the average generation number. In this experiment, the average number of successful runs for the paths is 96.91 and the standard deviation is 9.19. The formula, (number of successful runs – 96.91) / 9.19, computes the standard statistical Z score for each path's number of successful runs. The smallest Z score is -9.02 and the largest Z score is 0.34. There are 6 Z score ranges which are: $Z < -3$, $-3 \leq Z < -2$, $-2 \leq Z < -1$, $-1 \leq Z < 0$, $Z = 0$, $0 < Z \leq 1$. Count all of the paths whose Z score falls within each range. The chart in figure 6.46 illustrates the Z score counts for the crossover paths with the Z score ranges along the x-axis and the path count along the y-axis. The number of successful runs for a path

is an integer. Because the average number of successful runs per path is 96.91, a Z score greater than 0 is a path whose number of successful runs is between 97 and 100. From the chart in figure 6.46, 84.46% of the paths have successful runs between 97 and 100. 7,549 paths have 100 successful runs and 897 paths have successful runs between 97 and 99. Still, a path with $-3 \leq Z < 0$ is a path with a number of successful runs between 70 and 96. Path number 9573 is the path with the least number of successful runs of 14 having a Z score of -9.02. The next subsection analyzes the number of unsuccessful runs for the crossover paths.

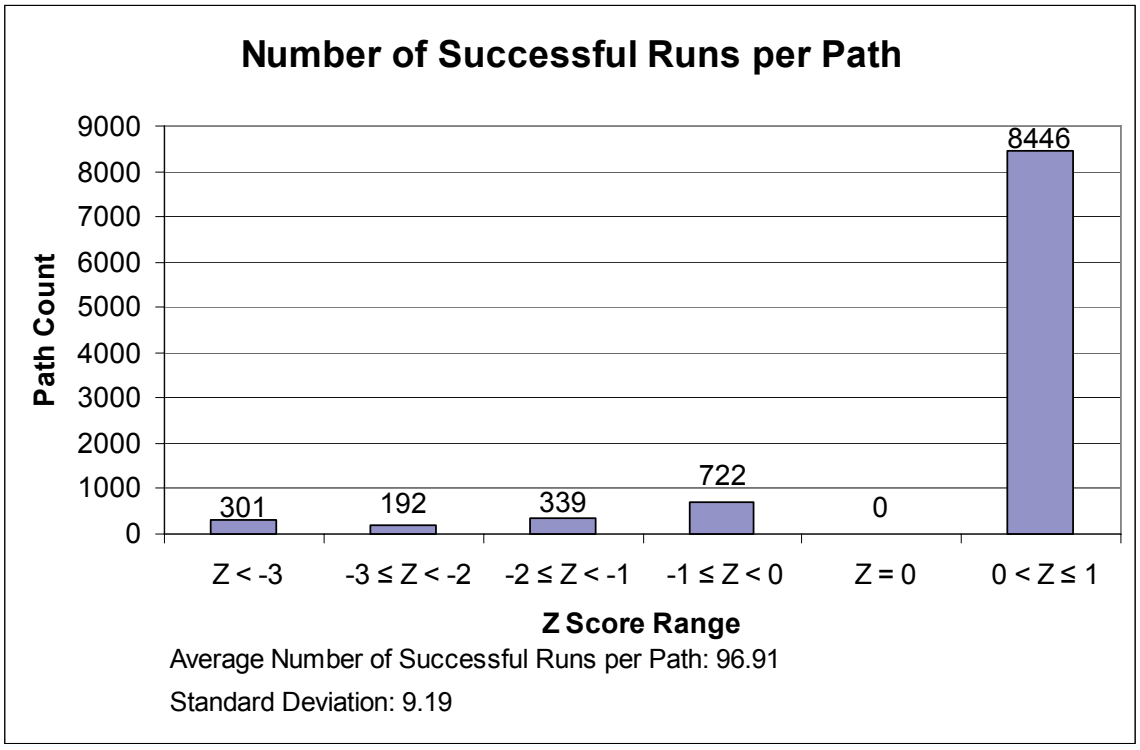


Figure 6.46 - Experiment 4's path counts for the number of successful runs per path.

6.1.6.2 Number of Unsuccessful Runs

After the fourth experiment processes all 10,000 crossover paths, a separate data processing program collects data about the number of unsuccessful runs for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length. The length of runCounts is 2645 with its indices ranging from 0 to 2644. The i^{th} element of runCounts corresponds to path length i. The program initializes each element in runCounts to zero. For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length.

The scatter chart in figure 6.47 illustrates the number of unsuccessful runs for the crossover paths with the number of unsuccessful runs along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the number of unsuccessful runs for a particular path length. The line at 0 unsuccessful runs is actually a series of points for a large number of path lengths. As expected, the number of unsuccessful runs increases in tandem with the path length. A comparison of the chart of unsuccessful runs for the fourth experiment to the charts of unsuccessful runs for the first three experiments reveals the fourth experiment has, on average, twelve times more unsuccessful runs than each of the first three experiments. Therefore, the agents have a much harder time recognizing the crossover path's geometry than the non-crossover path's geometry. The next subsection analyzes the average final generation number for the crossover paths.

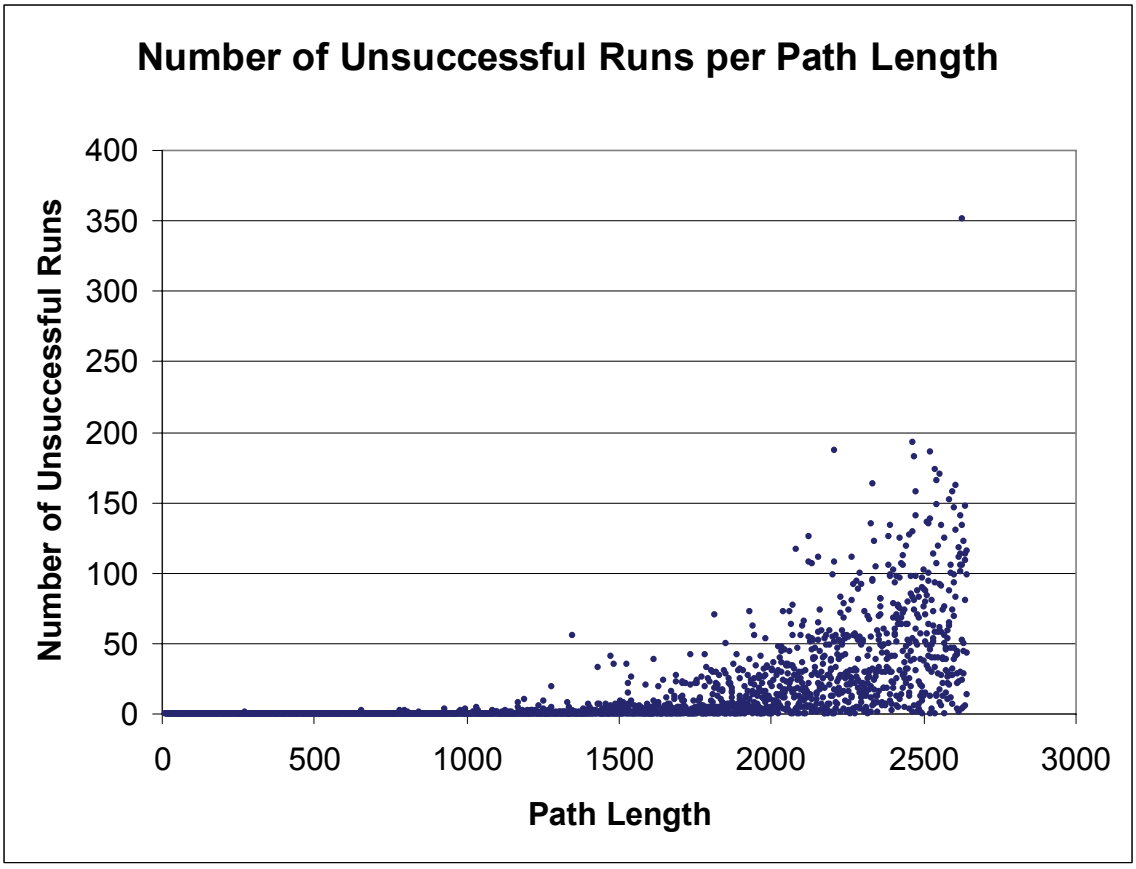


Figure 6.47 - The number of unsuccessful runs per path length for experiment 4.

6.1.6.3 Average Final Generation Number

After the fourth experiment processes all 10,000 crossover paths, a separate data processing program collects data about the average final generation number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of successful runs for each unique path length as well as another one dimensional array, called generationSums, to store the summation of the final generation numbers for the successful runs of each unique path length. The length of each array, runCounts and

generationSums, is 2645 with their indices ranging from 0 to 2644. The i^{th} element of each array, runCounts and generationSums, corresponds to path length i. The program initializes each element in runCounts and generationSums to zero.

For each path, the program retrieves the path's length and computes the number of successful runs for that path, adding the number of successful runs to the element in runCounts corresponding to the path's length. For each successful run of a path, the program retrieves the path's length and the final generation number of the successful run, adding the final generation number to the element in generationSums corresponding to the path's length. After the data processing program processes all 10,000 crossover paths, the program computes the average final generation number of each path length i as generationSums[i] / runCounts[i], where runCounts[i] > 0.

The scatter chart in figure 6.48 illustrates the average final generation number for the crossover paths with the average final generation number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average final generation number for a particular path length. As expected, the final generation number increases in tandem with the path length. The final generation numbers fall within a narrow range for the path length range of 13 to approximately 1200 meaning the path geometry for neighboring path lengths, within this range, remains consistent such that the signature recognition genetic algorithm requires the same amount of work to find agents to recognize those paths. For paths whose length is above 1200, the final generation numbers fall within a larger range meaning the path geometry for neighboring path lengths varies greatly such that the signature recognition genetic algorithm requires varying degrees of work to find agents to recognize those paths. For paths whose length

is above 1500, the ranges of final generation numbers for the path lengths continue to follow the curve of the ranges of final generation numbers for the path lengths below 1500 which is not the case in experiments one through three. The next subsection examines the agents to determine if any of them are contiguous agents.

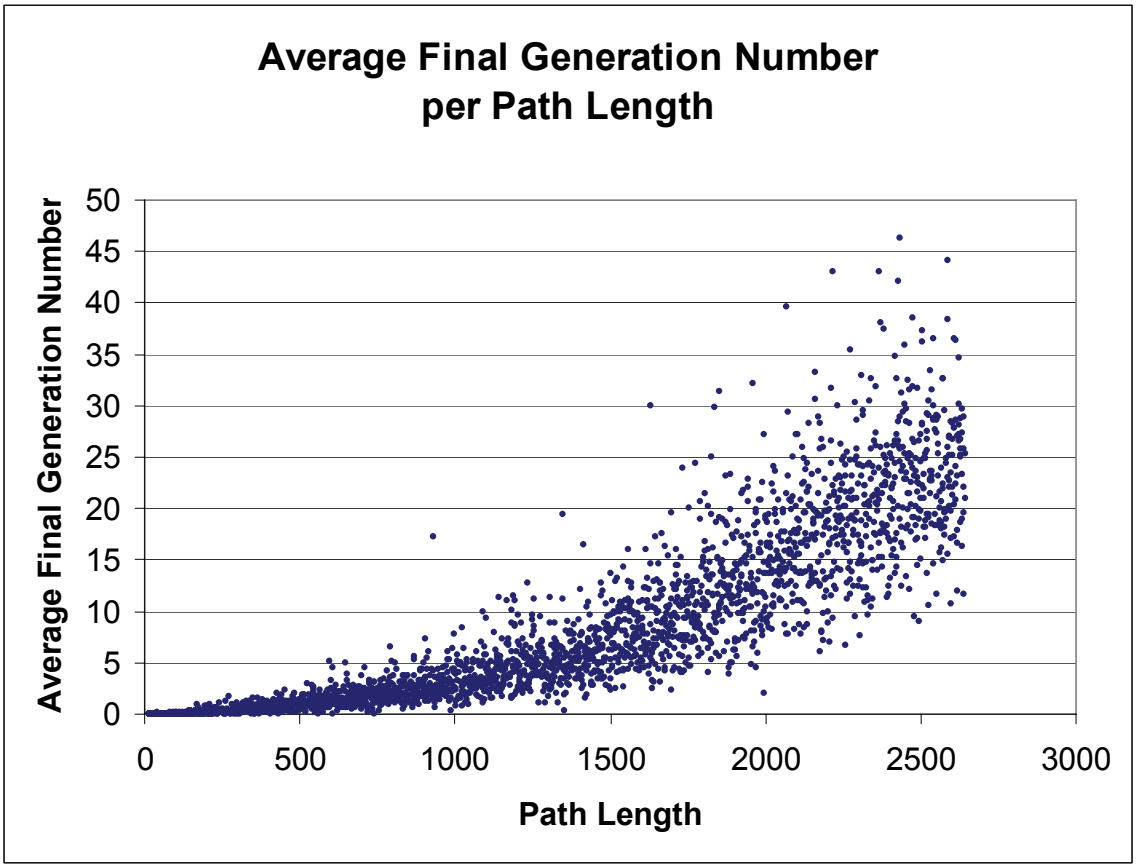


Figure 6.48 - The average final generation number per path length for experiment 4.

6.1.6.4 Contiguous Agents

Determine if an agent written to the fourth experiment's output file consumes its path in full contiguous order from beginning to end? A full contiguous order means the agent consumes each path square in order from the beginning path square to the end path

square. An agent can also consume the path in a partial contiguous order. A partial contiguous order means the agent consumes segments of the path in a full contiguous order but the agent consumes the path squares between segments out of order. The agent consumes path squares within a segment in a full contiguous order.

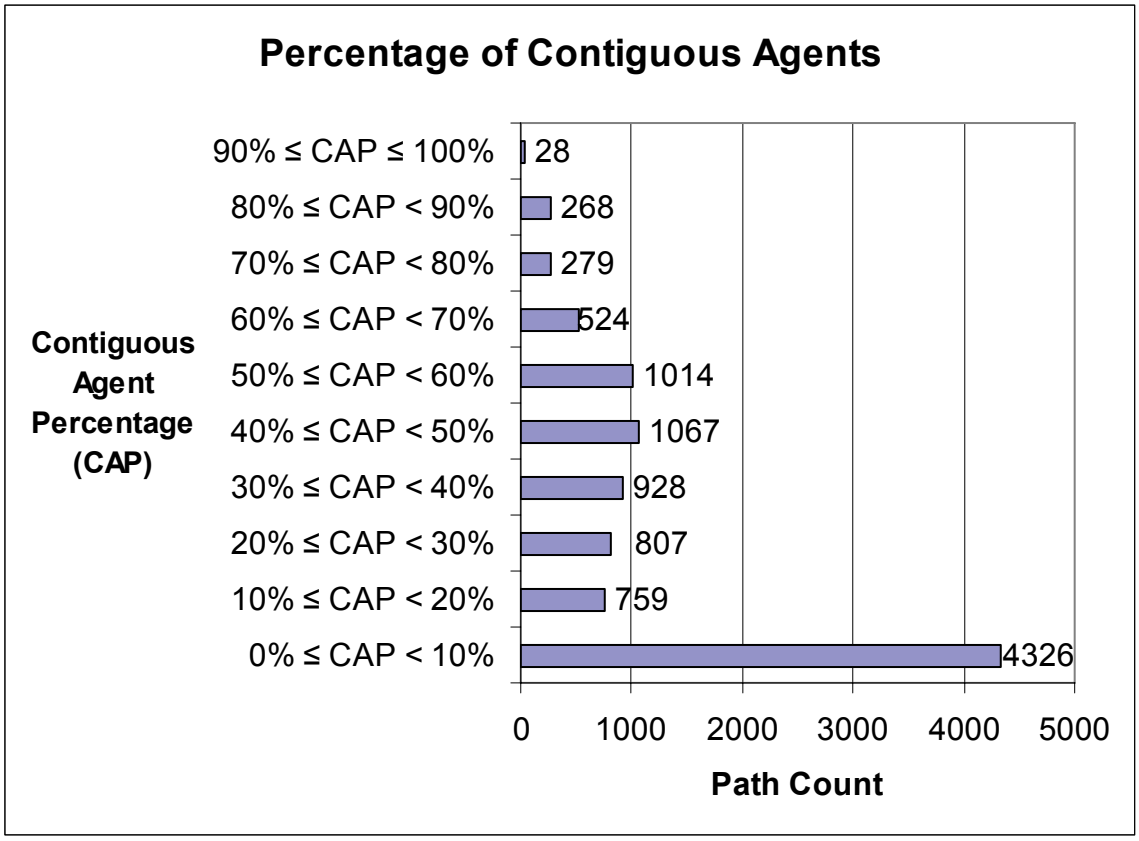


Figure 6.49 - Experiment 4's path counts for percentage of contiguous agents.

Count all of the agents written to the output file for a path. The fitness function computeFSMPCFitness, section 2.3 in chapter 2, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. The function computeFSMPCFitness can determine if an agent consumes a path in full contiguous order because full contiguous order is a special case of partial contiguous

order. The fitness value returned by computeFSMPCFitness is the number of path squares consumed in a partial contiguous order. Therefore, if an agent's fitness value computed by computeFSMPCFitness is equal to the path's length then the agent consumed the path in full contiguous order; hence, the agent is a contiguous agent. Count all of the path's contiguous agents. Divide the path's number of contiguous agents by the path's total number of agents, multiplied by 100 to compute the path's percentage of contiguous agents.

The contiguous agent percentage (CAP) runs from 0% to 100% divided into 10 ranges which are: $0\% \leq CAP < 10\%$, $10\% \leq CAP < 20\%$, $20\% \leq CAP < 30\%$, $30\% \leq CAP < 40\%$, $40\% \leq CAP < 50\%$, $50\% \leq CAP < 60\%$, $60\% \leq CAP < 70\%$, $70\% \leq CAP < 80\%$, $80\% \leq CAP < 90\%$, $90\% \leq CAP \leq 100\%$. Count all of the paths whose contiguous agent percentage falls within each range. The chart in figure 6.49 illustrates the path count for the contiguous agent percentage with the contiguous agent percentage ranges along the y-axis and the path count along the x-axis. Unlike the contiguous agent charts from experiment's one through three, the range $0\% \leq CAP < 10\%$ for experiment four has a large count. An examination of the contiguous agent data reveals that 3,378 paths have 0 contiguous agents with the remaining 948 paths in the range $0\% \leq CAP < 10\%$ having at least 1 contiguous agent. Removal of the 3,378 zero contiguous agent paths from the chart of the path counts produces a chart forming a bell curve with the top of the curve at the 40% to 50% range indicating a well distributed set of paths. The results of the fourth experiment demonstrate that the structure of the crossover paths is more difficult for the agents to recognize than the non-crossover paths. The next subsection examines the results of the fifth and last signature recognition program experiment.

6.1.7 The Fifth Signature Recognition Program Experiment

The examination of the fifth signature recognition program experiment begins in the next subsection with an analysis of the number of successful runs for the crossover paths. Subsection 6.1.7.2 analyzes the number of unsuccessful runs for the crossover paths. Subsection 6.1.7.3 studies the average final generation numbers for the crossover paths. Recall from section 6.1, this experiment uses the computeFSMPCFitness fitness function in which the order of path consumption is important, meaning that a successful run produces agents that consume the entire path in full contiguous order. Also, for an unsuccessful run of the genetic algorithm, this experiment writes the agent with the highest fitness value to the output file, meaning the agent consumes a portion of the path in a strict partial contiguous order. A k-segment is the largest segment of the path recognized, in a full contiguous order, by the agent. The computeFSMPCFitness fitness function keeps track of the length of the agent's k-segment for the current path in the grid. Subsection 6.1.7.4 examines the average k-segment numbers for the crossover paths. Finally, subsection 6.1.7.5 surveys the agents found by the signature recognition program for the unsuccessful runs of the fifth experiment.

6.1.7.1 Number of Successful Runs

For this experiment, the signature recognition program completes 100 runs of the genetic algorithm for each path. After processing all 10,000 paths, a separate data processing program collects data about each path, from the output files, computing the following four data values for each path: the number of successful runs of the path, the

average number of agents per run, the average number of reachable states per agent, and the average generation number. In this experiment, the average number of successful runs for the paths is 73.53 and the standard deviation is 41.27. The formula, (number of successful runs – 73.53) / 41.27, computes the standard statistical Z score for each path's number of successful runs. The smallest Z score is -1.78 and the largest Z score is 0.64. There are 6 Z score ranges which are: $Z < -3$, $-3 \leq Z < -2$, $-2 \leq Z < -1$, $-1 \leq Z < 0$, $Z = 0$, $0 < Z \leq 1$. Count all of the paths whose Z score falls within each range.

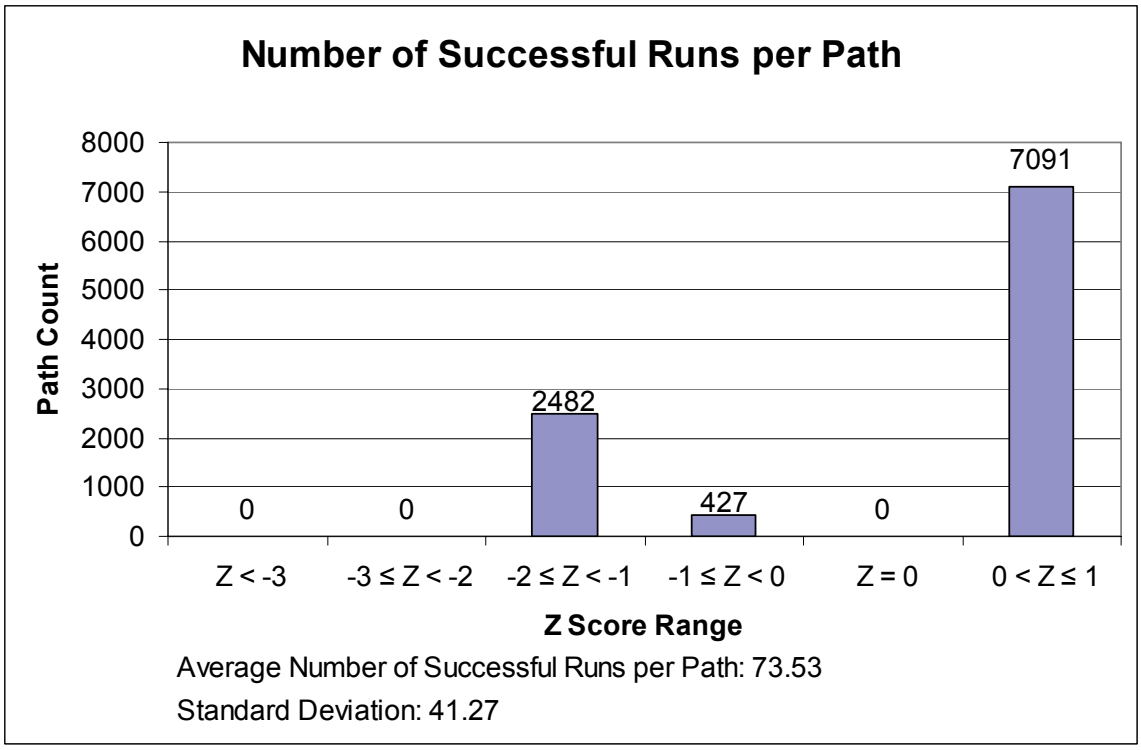


Figure 6.50 - Experiment 5's path counts for the number of successful runs per path.

The chart in figure 6.50 illustrates the Z score counts for the crossover paths with the Z score ranges along the x-axis and the path count along the y-axis. The number of successful runs for a path is an integer. Because the average number of successful runs per path is 73.53, a Z score greater than 0 is a path whose number of successful runs is

between 74 and 100. From the chart in figure 6.50, 70.91% of the paths have successful runs between 74 and 100. 6,201 paths have 100 successful runs and 890 paths have successful runs between 74 and 99. A path with $-2 \leq Z < -1$ is a path with a number of successful runs between 0 and 32. 1,446 paths have 0 successful runs and 1,036 paths have successful runs between 1 and 32. A path with $-1 \leq Z < 0$ is a path with a number of successful runs between 33 and 73. The next subsection analyzes the number of unsuccessful runs for the crossover paths.

6.1.7.2 Number of Unsuccessful Runs

After the fifth experiment processes all 10,000 crossover paths, a separate data processing program collects data about the number of unsuccessful runs for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length. The length of runCounts is 2645 with its indices ranging from 0 to 2644. The i^{th} element of runCounts corresponds to path length i. The program initializes each element in runCounts to zero. For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length.

The scatter chart in figure 6.51 illustrates the number of unsuccessful runs for the crossover paths with the number of unsuccessful runs along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the number of unsuccessful runs for a particular path length. The line at 0 unsuccessful runs is actually a series of points

for a large number of path lengths. As expected, the number of unsuccessful runs increases in tandem with the path length. A comparison of the chart of unsuccessful runs for the fifth experiment to the charts of unsuccessful runs for the first three experiments reveals the fifth experiment has, on average, 103 times more unsuccessful runs than each of the first three experiments. A comparison of the chart of unsuccessful runs for the fifth experiment to the chart of unsuccessful runs for the fourth experiment reveals the fifth experiment has nine times more unsuccessful runs than the fourth experiment.

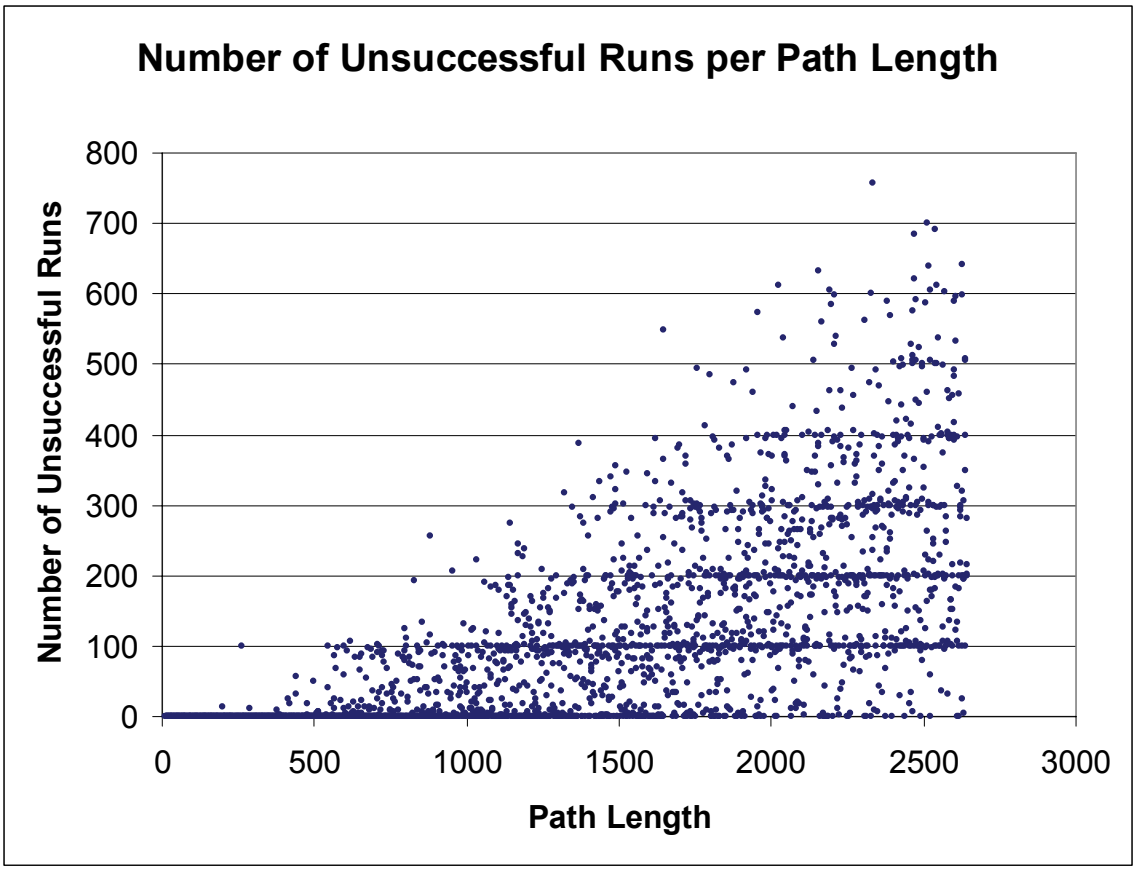


Figure 6.51 - The number of unsuccessful runs per path length for experiment 5.

The order of path consumption is immaterial for the fitness calculation of the agents in experiments one through four while the fitness calculation in experiment five is

more difficult because the order of path consumption is important. Therefore, the agents of experiment five have a much harder time recognizing the crossover paths than in experiment four. Also, the agents of experiment five have a much harder time recognizing the crossover paths than the agents recognizing the non-crossover paths in experiments one through three because of the stricter fitness computation and the complex geometry of the crossover paths as compared to the simpler geometry of the non-crossover paths. The next subsection analyzes the average final generation number for the crossover paths.

6.1.7.3 Average Final Generation Number

After the fifth experiment processes all 10,000 crossover paths, a separate data processing program collects data about the average final generation number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. The program maintains a one dimensional array, called runCounts, to store the number of successful runs for each unique path length as well as another one dimensional array, called generationSums, to store the summation of the final generation numbers for the successful runs of each unique path length. The length of each array, runCounts and generationSums, is 2645 with their indices ranging from 0 to 2644. The i^{th} element of each array, runCounts and generationSums, corresponds to path length i. The program initializes each element in runCounts and generationSums to zero.

For each path, the program retrieves the path's length and computes the number of successful runs for that path, adding the number of successful runs to the element in runCounts corresponding to the path's length. For each successful run of a path, the

program retrieves the path's length and the final generation number of the successful run, adding the final generation number to the element in generationSums corresponding to the path's length. After the data processing program processes all 10,000 crossover paths, the program computes the average final generation number of each path length i as generationSums[i] / runCounts[i], where runCounts[i] > 0.

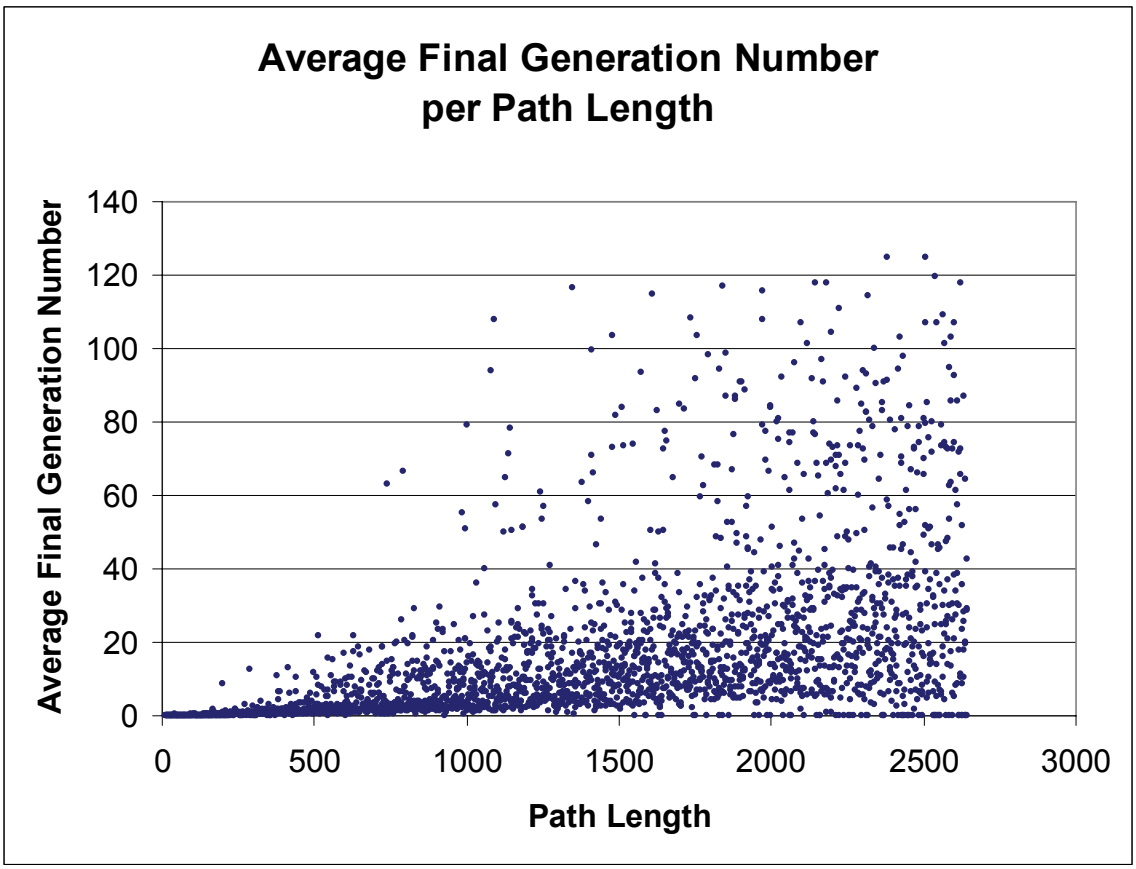


Figure 6.52 - The average final generation number per path length for experiment 5.

The scatter chart in figure 6.52 illustrates the average final generation number for the crossover paths with the average final generation number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average final generation number for a particular path length. As expected, the final generation number

increases in tandem with the path length. Unlike experiments one through four, the final generation numbers for the fifth experiment spread throughout the entire range of generations from generation 0 to generation 125 except for the paths whose length is less than 1000, where the final generation numbers fall within a narrower range. The higher final generation numbers of experiment five result from the experiment's difficult fitness calculation and the complex geometry of the crossover paths. The next subsection studies path segment lengths recognized by the agents found for the unsuccessful runs of this experiment.

6.1.7.4 Average K-Segment Number

After the fifth experiment processes all 10,000 crossover paths, a separate data processing program collects data about the average maximum length k-segment number for the paths. The path lengths range from the smallest path length of 13 to the largest path length of 2644. For an unsuccessful run of the genetic algorithm, this experiment writes the first agent with the highest fitness value to the output file, meaning the agent consumes a portion of the path in a strict partial contiguous order. A k-segment is the largest segment of the path recognized, in a full contiguous order, by the agent. The computeFSMPCFitness fitness function keeps track of the length of the agent's k-segment for the current path in the grid. The program maintains a one dimensional array, called runCounts, to store the number of unsuccessful runs for each unique path length as well as another one dimensional array, called ksegmentSums, to store the summation of the k-segment numbers for the unsuccessful runs of each unique path length. The length of each array, runCounts and ksegmentSums, is 2645 with their indices ranging from 0 to

2644. The i^{th} element of each array, runCounts and ksegmentSums, corresponds to path length i. The program initializes each element in runCounts and ksegmentSums to zero.

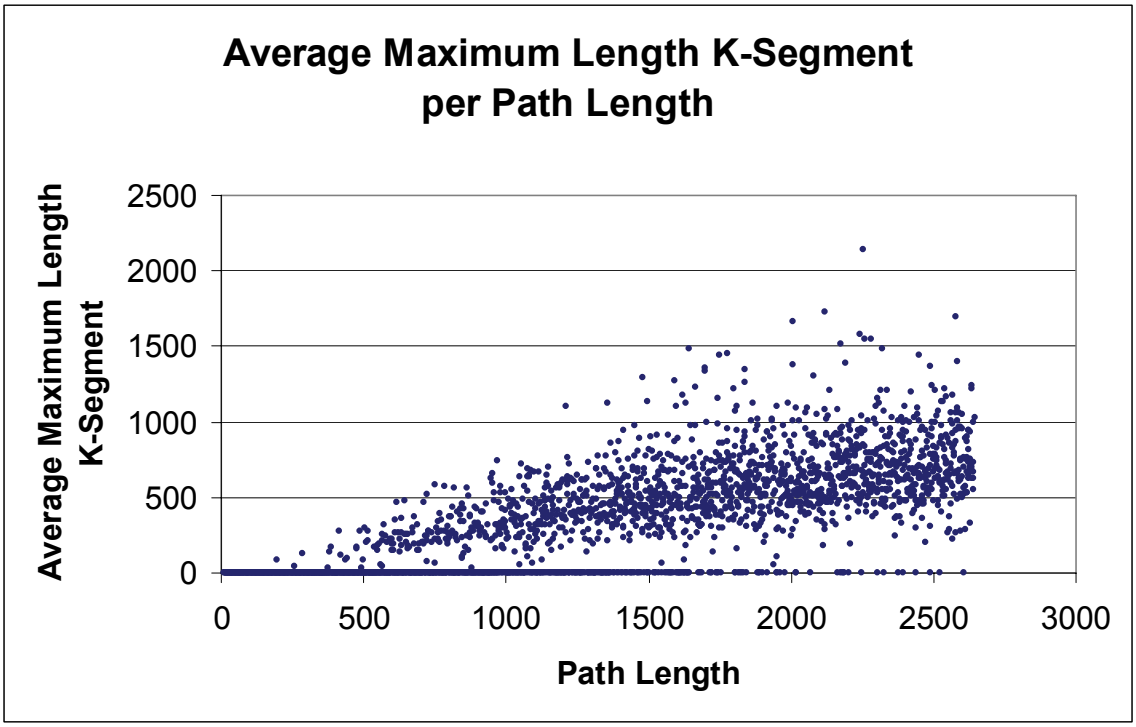


Figure 6.53 - The average maximum length k-segment per path length for experiment 5.

For each path, the program retrieves the path's length and computes the number of unsuccessful runs for that path, adding the number of unsuccessful runs to the element in runCounts corresponding to the path's length. For each unsuccessful run of a path, the program retrieves the path's length and the agent of the unsuccessful run. The program executes the computeFSMPCFitness fitness function on the agent for the path. When the fitness function completes execution, the program retrieves the agent's k-segment number for the path, adding the k-segment number to the element in ksegmentSums corresponding to the path's length. After the data processing program processes all

10,000 crossover paths, the program computes the average k-segment number of each path length i as ksegmentSums[i] / runCounts[i], where runCounts[i] > 0.

The scatter chart in figure 6.53 illustrates the average k-segment number for the crossover paths with the average k-segment number along the y-axis and the path length along the x-axis. Each point in the chart corresponds to the average k-segment number for a particular path length. The line at 0 k-segment number is actually a series of points for a large number of path lengths, paths having 100 successful runs. As expected, the k-segment number increases in tandem with the path length. The k-segment numbers fall within a narrow range for the entire path length range with the size of the k-segments as 30%, on average, of the path's length. The next subsection examines the agents found for the unsuccessful runs of this experiment.

6.1.7.5 Partial Contiguous Agents

The fitness function computeFSMPCFitness, section 2.3 in chapter 2, measures how much of the path the agent consumes but only those path squares consumed in a partial contiguous order. The fitness value returned by computeFSMPCFitness is the number of path squares consumed in a partial contiguous order. For an unsuccessful run of the genetic algorithm, this experiment writes the agent with the highest fitness value to the output file. For all unsuccessful runs of a path, compute the fitness value of the agent for the path. Divide the fitness value by the path's length to get the percentage of the path the agent consumed in a partial contiguous order. Compute the average partial contiguous fitness percentage for the path.

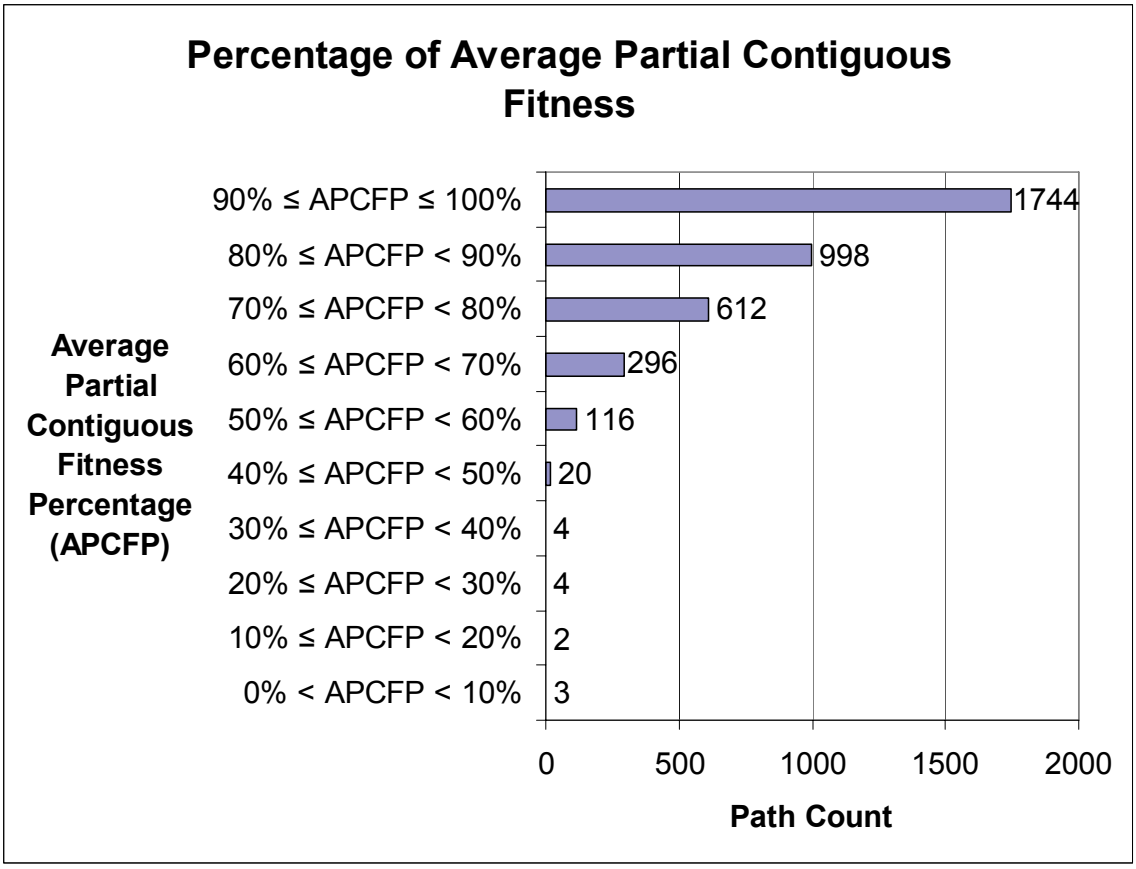


Figure 6.54 - Experiment 5's path counts of average partial contiguous fitness percentage.

The average partial contiguous fitness percentage (APCFP) runs from 0% to 100% divided into 10 ranges which are: 0% ≤ APCFP < 10%, 10% ≤ APCFP < 20%, 20% ≤ APCFP < 30%, 30% ≤ APCFP < 40%, 40% ≤ APCFP < 50%, 50% ≤ APCFP < 60%, 60% ≤ APCFP < 70%, 70% ≤ APCFP < 80%, 80% ≤ APCFP < 90%, 90% ≤ APCFP ≤ 100%. Count all of the paths whose average partial contiguous fitness percentage falls within each range. The chart in figure 6.54 illustrates the path counts for the crossover paths with the average partial contiguous fitness percentage ranges along the y-axis and the path counts along the x-axis. An examination of the results of the fifth signature recognition program experiment data reveals that 3,799 paths have at least 1

unsuccessful run. The chart of the path counts forms an exponential curve with most of the paths in the 60% to 100% range indicating that the agents consumed a large part of the path in a partial contiguous order. The results of this experiment as well as the fourth experiment demonstrate that the structure of the crossover paths is more difficult, but not impossible, for the agents to recognize than the non-crossover paths. The next section discusses the results of the two experiments of the optical character recognition problem described in chapter 3.

6.2 Optical Character Recognition Program

The line recognition genetic algorithm, section 3.4 in chapter 3, is a major component of the optical character recognition and character database creation programs, section 3.6 in chapter 3. Given a page of text, the line recognition genetic algorithm determines the number of lines of text on the page. The next subsection discusses the results of the experiments to test the line recognition genetic algorithm. The character recognition phase of the optical character recognition program uses the character database created by the character database creation program. Subsection 6.2.2 provides a study of the measure and moment values of the characters in the database.

6.2.1 Line Recognition Genetic Algorithm Experiments

Eight pages of text, acquired and scanned, are the non-skewed version of the test pages. Even though they are “non-skewed”, actually, they are askew by a very small degree. The layout of the text on the first five pages follows a portrait orientation. The layout of the text on the remaining three pages follows a landscape orientation. The

pages of text test the line recognition genetic algorithm under the following conditions. A detailed description of the formatting on a text page is in the paragraph dedicated to that particular text page. The first text page has consistent font name and font size with few blank spaces between lines. The second text page has two different font names and three different font sizes with varying degrees of blank spaces between groups of lines. The third text page uses a single font name but two different font sizes with varying degrees of blank spaces between groups of lines. The fourth text page uses a single font name but four different font sizes scattered through the page with varying degrees of blank spaces between groups of lines. The fifth text page tests the line recognition genetic algorithm on a copy of an original page of text with a number of stray marks. The sixth text page is similar to the third text page but with a landscape orientation. The seventh text page is a landscape oriented text page similar to the sixth text page but with more blank spaces between groups of lines. The eighth and final text page just tests the line recognition genetic algorithm for a uniform large size text at a landscape orientation.

Skew each page of text by the four angles: 6° counter-clockwise, 2° counter-clockwise, 2° clockwise and 6° clockwise to create four additional versions of each text page. Therefore, a total of 40 test pages tests the performance of the line recognition genetic algorithm. After the line recognition genetic algorithm finds the text lines in the test page, the optical character recognition program draws, in the image of the test page, a line at each text line's proposed position. The optical character recognition program draws the resulting image to a new file. A separate figure displays the resulting "lined" image of each test page.

The first test page is the first text page not skewed and is in figure 6.55. The Times New Roman font and font size of 12 formats the text in the page. Most of the text is normal with several words bolded. The first text page skewed 6° counter-clockwise is in figure 6.56. The first text page skewed 2° counter-clockwise is in figure 6.57. The first text page skewed 2° clockwise is in figure 6.58. The first text page skewed 6° clockwise is in figure 6.59. There are 43 lines of text in these five test pages. Reviewing the first four test pages shows that the line recognition genetic algorithm computes exactly 43 lines of text in each of these four test pages. But, the fifth test page in figure 6.59 shows that the line recognition genetic algorithm computes 44 lines of text, one line of text has two lines drawn through the line of text.

The sixth test page is the second text page not skewed and is in figure 6.60. The Times New Roman font and font size of 12 formats most of the text in the page. The Times New Roman font and font size of 18 formats the first line. The Times New Roman font and font size of 14 formats the second line. The Courier New font formats the web address and letter grade ranges on the page. Most of the text is normal with the first two lines bolded. The second text page skewed 6° counter-clockwise is in figure 6.61. The second text page skewed 2° counter-clockwise is in figure 6.62. The second text page skewed 2° clockwise is in figure 6.63. The second text page skewed 6° clockwise is in figure 6.64. There are 29 lines of text in these five test pages. Reviewing all five test pages shows that the line recognition genetic algorithm computes exactly 29 lines of text in each of these five test pages. The thin elongated triangle shaped object at the bottom of the test pages in figures 6.60, 6.63 and 6.64 is a scan artifact, a scan area

not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

The eleventh test page is the third text page not skewed and is in figure 6.65. The Times New Roman font and font size of 11 formats most of the text in the page. The Times New Roman font and font size of 16 formats the first and last lines. The third text page skewed 6° counter-clockwise is in figure 6.66. The third text page skewed 2° counter-clockwise is in figure 6.67. The third text page skewed 2° clockwise is in figure 6.68. The third text page skewed 6° clockwise is in figure 6.69. There are 37 lines of text in these five test pages. Reviewing all five test pages shows that the line recognition genetic algorithm computes exactly 37 lines of text in each of these five test pages. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

The sixteenth test page is the fourth text page not skewed and is in figure 6.70. The Times New Roman font and a mixture of the font sizes 11, 14, 16 and 18 format the text in the page. The fourth text page skewed 6° counter-clockwise is in figure 6.71. The fourth text page skewed 2° counter-clockwise is in figure 6.72. The fourth text page skewed 2° clockwise is in figure 6.73. The fourth text page skewed 6° clockwise is in figure 6.74. There are 23 lines of text in these five test pages. Reviewing all five test pages shows that the line recognition genetic algorithm computes exactly 23 lines of text in each of these five test pages. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

The twenty-first test page is the fifth text page not skewed and is in figure 6.75. The page of text is a set of instructions from a dental office. The font and font size are consistent throughout the page. The fifth text page skewed 6° counter-clockwise is in figure 6.76. The fifth text page skewed 2° counter-clockwise is in figure 6.77. The fifth text page skewed 2° clockwise is in figure 6.78. The fifth text page skewed 6° clockwise is in figure 6.79. There are 28 lines of text in these five test pages. Reviewing the three test pages, in figures 6.75, 6.76 and 6.77, shows that the line recognition genetic algorithm computes exactly 28 lines of text in each of these three test pages. But, the line recognition genetic algorithm incorrectly detected an extra 29^{th} line at the second line of text in the clockwise skewed test pages of figures 6.78 and 6.79. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. In addition, there are specks of dirt in the image. The line recognition genetic algorithm correctly ignores all of these artifacts by design.

The twenty-sixth test page is the sixth text page not skewed and is in figure 6.80. The Times New Roman font and font size of 12 formats the text in the page. Most of the text is normal with several words bolded. This is the first of three text pages in which the layout of the text is landscape. The sixth text page skewed 6° counter-clockwise is in figure 6.81. The sixth text page skewed 2° counter-clockwise is in figure 6.82. The sixth text page skewed 2° clockwise is in figure 6.83. The sixth text page skewed 6° clockwise is in figure 6.84. There are 21 lines of text in these five test pages. Reviewing all five test pages shows that the line recognition genetic algorithm correctly detects the 21 lines of text on the page but incorrectly detects an extra line consisting of a large speck of dirt between the fifth and sixth lines, from the right side of the image. In addition, in the

counter-clockwise skewed test pages of figures 6.81 and 6.82, the line recognition genetic algorithm incorrectly detected an addition extra line, 23rd, due to a thin short artifact at the left side of the image. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

The thirty-first test page is the seventh text page not skewed and is in figure 6.85. The Times New Roman font and font size of 12 formats the text in the page. Most of the text is normal with numerous words bolded. This is the second text page in which the layout of the text is landscape. The seventh text page skewed 6° counter-clockwise is in figure 6.86. The seventh text page skewed 2° counter-clockwise is in figure 6.87. The seventh text page skewed 2° clockwise is in figure 6.88. The seventh text page skewed 6° clockwise is in figure 6.89. There are 20 lines of text in these five test pages. Reviewing all five test pages shows that the line recognition genetic algorithm computes exactly 20 lines of text in each of these five test pages. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

The thirty-sixth test page is the eighth text page not skewed and is in figure 6.90. The Georgia font and font size of 30 formats the text in the page. This is the third text page in which the layout of the text is landscape. The eighth text page skewed 6° counter-clockwise is in figure 6.91. The eighth text page skewed 2° counter-clockwise is in figure 6.92. The eighth text page skewed 2° clockwise is in figure 6.93. The eighth text page skewed 6° clockwise is in figure 6.94. There are 7 lines of text in these five test

pages. Reviewing all five test pages shows that the line recognition genetic algorithm computes exactly 7 lines of text in each of these five test pages. It is odd that the drawn lines appear not to go through the horizontal center of each text line in the not skewed eighth text page of figure 6.90. Though, this is not the case for the other four skewed eighth text pages. The thin elongated triangle shaped object at the bottom of the five test pages is a scan artifact, a scan area not covered by the paper. The line recognition genetic algorithm correctly ignores this artifact by design.

Eight pages of text, acquired and scanned, are the non-skewed version of the test pages. Skew each page of text by the four angles: 6° counter-clockwise, 2° counter-clockwise, 2° clockwise and 6° clockwise to create four additional versions of each text page. Therefore, a total of 40 test pages tests the performance of the line recognition genetic algorithm. For the most part, the line recognition genetic algorithm correctly recognized each line on the test page. Skew angle or page orientation did not adversely affect the outcome of the line recognition genetic algorithm. There were some cases where the line recognition genetic algorithm recognized two lines for one text line. In other cases, the line recognition genetic algorithm identified large specks of dirt or odd stray marks as potential characters recognizing them as a line or lines. Overall, the line recognition genetic algorithm performed well for the test pages. The next subsection provides a study of the measure and moment values of the characters in the character recognition database.

Section 1. – An Introduction to the Ant Experiment.

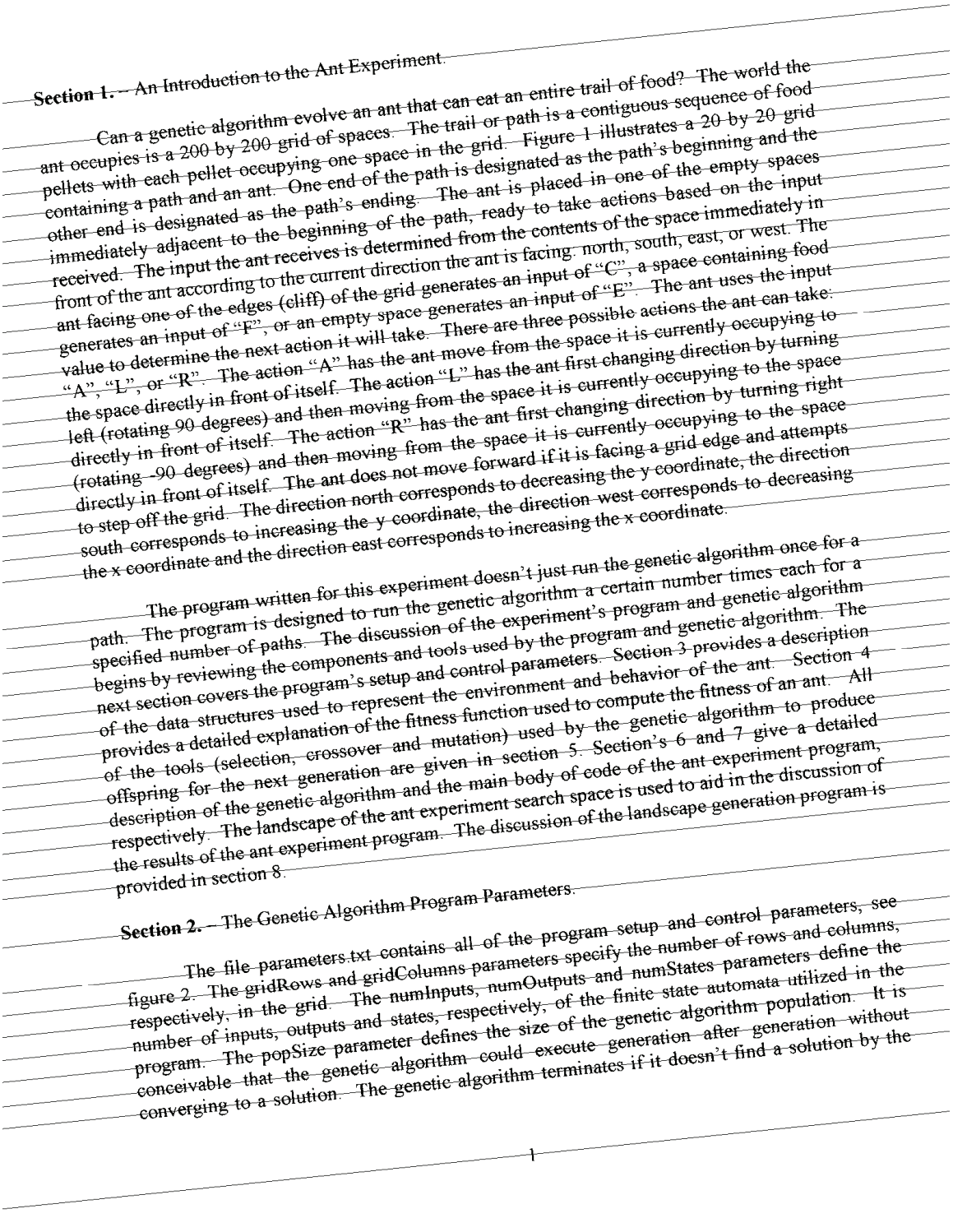
Can a genetic algorithm evolve an ant that can eat an entire trail of food? The world the ant occupies is a 200 by 200 grid of spaces. The trail or path is a contiguous sequence of food pellets with each pellet occupying one space in the grid. Figure 1 illustrates a 20 by 20 grid containing a path and an ant. One end of the path is designated as the path's beginning and the other end is designated as the path's ending. The ant is placed in one of the empty spaces immediately adjacent to the beginning of the path, ready to take actions based on the input received. The input the ant receives is determined from the contents of the space immediately in front of the ant according to the current direction the ant is facing: north, south, east, or west. The ant facing one of the edges (cliff) of the grid generates an input of "C", a space containing food generates an input of "F", or an empty space generates an input of "E". The ant uses the input value to determine the next action it will take. There are three possible actions the ant can take: "A", "L", or "R". The action "A" has the ant move from the space it is currently occupying to the space directly in front of itself. The action "L" has the ant first changing direction by turning left (rotating 90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The action "R" has the ant first changing direction by turning right (rotating -90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The ant does not move forward if it is facing a grid edge and attempts to step off the grid. The direction north corresponds to decreasing the y coordinate, the direction south corresponds to increasing the y coordinate, the direction west corresponds to decreasing the x coordinate and the direction east corresponds to increasing the x coordinate.

The program written for this experiment doesn't just run the genetic algorithm once for a path. The program is designed to run the genetic algorithm a certain number times each for a specified number of paths. The discussion of the experiment's program and genetic algorithm begins by reviewing the components and tools used by the program and genetic algorithm. The next section covers the program's setup and control parameters. Section 3 provides a description of the data structures used to represent the environment and behavior of the ant. Section 4 provides a detailed explanation of the fitness function used to compute the fitness of an ant. All of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation are given in section 5. Section's 6 and 7 give a detailed description of the genetic algorithm and the main body of code of the ant experiment program, respectively. The landscape of the ant experiment search space is used to aid in the discussion of the results of the ant experiment program. The discussion of the landscape generation program is provided in section 8.

Section 2. – The Genetic Algorithm Program Parameters.

The file parameters.txt contains all of the program setup and control parameters, see figure 2. The gridRows and gridColumns parameters specify the number of rows and columns, respectively, in the grid. The numInputs, numOutputs and numStates parameters define the number of inputs, outputs and states, respectively, of the finite state automata utilized in the program. The popSize parameter defines the size of the genetic algorithm population. It is conceivable that the genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm terminates if it doesn't find a solution by the

Figure 6.55 - First test page of text, not skewed.



Section 1. – An Introduction to the Ant Experiment.

Can a genetic algorithm evolve an ant that can eat an entire trail of food? The world the ant occupies is a 200 by 200 grid of spaces. The trail or path is a contiguous sequence of food pellets with each pellet occupying one space in the grid. Figure 1 illustrates a 20 by 20 grid containing a path and an ant. One end of the path is designated as the path's beginning and the other end is designated as the path's ending. The ant is placed in one of the empty spaces immediately adjacent to the beginning of the path, ready to take actions based on the input received. The input the ant receives is determined from the contents of the space immediately in front of the ant according to the current direction the ant is facing: north, south, east, or west. The ant facing one of the edges (cliff) of the grid generates an input of "C", a space containing food generates an input of "F", or an empty space generates an input of "E". The ant uses the input value to determine the next action it will take. There are three possible actions the ant can take: "A", "L", or "R". The action "A" has the ant move from the space it is currently occupying to the space directly in front of itself. The action "L" has the ant first changing direction by turning left (rotating 90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The action "R" has the ant first changing direction by turning right (rotating -90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The ant does not move forward if it is facing a grid edge and attempts to step off the grid. The direction north corresponds to decreasing the y coordinate, the direction south corresponds to increasing the y coordinate, the direction west corresponds to decreasing the x coordinate and the direction east corresponds to increasing the x coordinate.

The program written for this experiment doesn't just run the genetic algorithm once for a path. The program is designed to run the genetic algorithm a certain number times each for a specified number of paths. The discussion of the experiment's program and genetic algorithm begins by reviewing the components and tools used by the program and genetic algorithm. The next section covers the program's setup and control parameters. Section 3 provides a description of the data structures used to represent the environment and behavior of the ant. Section 4 provides a detailed explanation of the fitness function used to compute the fitness of an ant. All of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation are given in section 5. Section's 6 and 7 give a detailed description of the genetic algorithm and the main body of code of the ant experiment program, respectively. The landscape of the ant experiment search space is used to aid in the discussion of the results of the ant experiment program. The discussion of the landscape generation program is provided in section 8.

Section 2. – The Genetic Algorithm Program Parameters.

The file parameters.txt contains all of the program setup and control parameters, see figure 2. The gridRows and gridColumns parameters specify the number of rows and columns, respectively, in the grid. The numInputs, numOutputs and numStates parameters define the number of inputs, outputs and states, respectively, of the finite state automata utilized in the program. The popSize parameter defines the size of the genetic algorithm population. It is conceivable that the genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm terminates if it doesn't find a solution by the

Figure 6.56 - First test page of text, skewed 6° counter-clockwise.

Section 1. – An Introduction to the Ant Experiment.

Can a genetic algorithm evolve an ant that can eat an entire trail of food? The world the ant occupies is a 200 by 200 grid of spaces. The trail or path is a contiguous sequence of food pellets with each pellet occupying one space in the grid. Figure 1 illustrates a 20 by 20 grid containing a path and an ant. One end of the path is designated as the path's beginning and the other end is designated as the path's ending. The ant is placed in one of the empty spaces immediately adjacent to the beginning of the path, ready to take actions based on the input received. The input the ant receives is determined from the contents of the space immediately in front of the ant according to the current direction the ant is facing: north, south, east, or west. The ant facing one of the edges (cliff) of the grid generates an input of "C", a space containing food generates an input of "F", or an empty space generates an input of "E". The ant uses the input value to determine the next action it will take. There are three possible actions the ant can take: "A", "L", or "R". The action "A" has the ant move from the space it is currently occupying to the space directly in front of itself. The action "L" has the ant first changing direction by turning left (rotating 90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The action "R" has the ant first changing direction by turning right (rotating -90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The ant does not move forward if it is facing a grid edge and attempts to step off the grid. The direction north corresponds to decreasing the y coordinate, the direction south corresponds to increasing the y coordinate, the direction west corresponds to decreasing the x coordinate and the direction east corresponds to increasing the x coordinate.

The program written for this experiment doesn't just run the genetic algorithm once for a path. The program is designed to run the genetic algorithm a certain number times each for a specified number of paths. The discussion of the experiment's program and genetic algorithm begins by reviewing the components and tools used by the program and genetic algorithm. The next section covers the program's setup and control parameters. Section 3 provides a description of the data structures used to represent the environment and behavior of the ant. Section 4 provides a detailed explanation of the fitness function used to compute the fitness of an ant. All of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation are given in section 5. Section's 6 and 7 give a detailed description of the genetic algorithm and the main body of code of the ant experiment program, respectively. The landscape of the ant experiment search space is used to aid in the discussion of the results of the ant experiment program. The discussion of the landscape generation program is provided in section 8.

Section 2. – The Genetic Algorithm Program Parameters.

The file parameters.txt contains all of the program setup and control parameters, see figure 2. The gridRows and gridColumns parameters specify the number of rows and columns, respectively, in the grid. The numInputs, numOutputs and numStates parameters define the number of inputs, outputs and states, respectively, of the finite state automata utilized in the program. The popSize parameter defines the size of the genetic algorithm population. It is conceivable that the genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm terminates if it doesn't find a solution by the

Figure 6.57 - First test page of text, skewed 2° counter-clockwise.

Section 1. – An Introduction to the Ant Experiment.

Can a genetic algorithm evolve an ant that can eat an entire trail of food? The world the ant occupies is a 200 by 200 grid of spaces. The trail or path is a contiguous sequence of food pellets with each pellet occupying one space in the grid. Figure 1 illustrates a 20 by 20 grid containing a path and an ant. One end of the path is designated as the path's beginning and the other end is designated as the path's ending. The ant is placed in one of the empty spaces immediately adjacent to the beginning of the path, ready to take actions based on the input received. The input the ant receives is determined from the contents of the space immediately in front of the ant according to the current direction the ant is facing: north, south, east, or west. The ant facing one of the edges (cliff) of the grid generates an input of "C", a space containing food generates an input of "F", or an empty space generates an input of "E". The ant uses the input value to determine the next action it will take. There are three possible actions the ant can take: "A", "L", or "R". The action "A" has the ant move from the space it is currently occupying to the space directly in front of itself. The action "L" has the ant first changing direction by turning left (rotating 90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The action "R" has the ant first changing direction by turning right (rotating -90 degrees) and then moving from the space it is currently occupying to the space directly in front of itself. The ant does not move forward if it is facing a grid edge and attempts to step off the grid. The direction north corresponds to decreasing the y coordinate, the direction south corresponds to increasing the y coordinate, the direction west corresponds to decreasing the x coordinate and the direction east corresponds to increasing the x coordinate.

The program written for this experiment doesn't just run the genetic algorithm once for a path. The program is designed to run the genetic algorithm a certain number times each for a specified number of paths. The discussion of the experiment's program and genetic algorithm begins by reviewing the components and tools used by the program and genetic algorithm. The next section covers the program's setup and control parameters. Section 3 provides a description of the data structures used to represent the environment and behavior of the ant. Section 4 provides a detailed explanation of the fitness function used to compute the fitness of an ant. All of the tools (selection, crossover and mutation) used by the genetic algorithm to produce offspring for the next generation are given in section 5. Section's 6 and 7 give a detailed description of the genetic algorithm and the main body of code of the ant experiment program, respectively. The landscape of the ant experiment search space is used to aid in the discussion of the results of the ant experiment program. The discussion of the landscape generation program is provided in section 8.

Section 2. – The Genetic Algorithm Program Parameters.

The file parameters.txt contains all of the program setup and control parameters, see figure 2. The gridRows and gridColumns parameters specify the number of rows and columns, respectively, in the grid. The numInputs, numOutputs and numStates parameters define the number of inputs, outputs and states, respectively, of the finite state automata utilized in the program. The popSize parameter defines the size of the genetic algorithm population. It is conceivable that the genetic algorithm could execute generation after generation without converging to a solution. The genetic algorithm terminates if it doesn't find a solution by the

Figure 6.58 - First test page of text, skewed 2° clockwise.

CS 12 Lab - Understanding and Using Personal Computers

Mr. Svitak

E-mail: cs12svitak@yahoo.com
Mailbox: SB A-201
Office: SB A-201
Office Hour: Monday, 12:30 – 1:30 p.m.

Web address for the CS 12 lab home page:
`http://eniac.cs.qc.cuny.edu/~svitak/cs12`

You will need to hand in to me a new blank IBM or PC formatted diskette for the 3 lab exams.

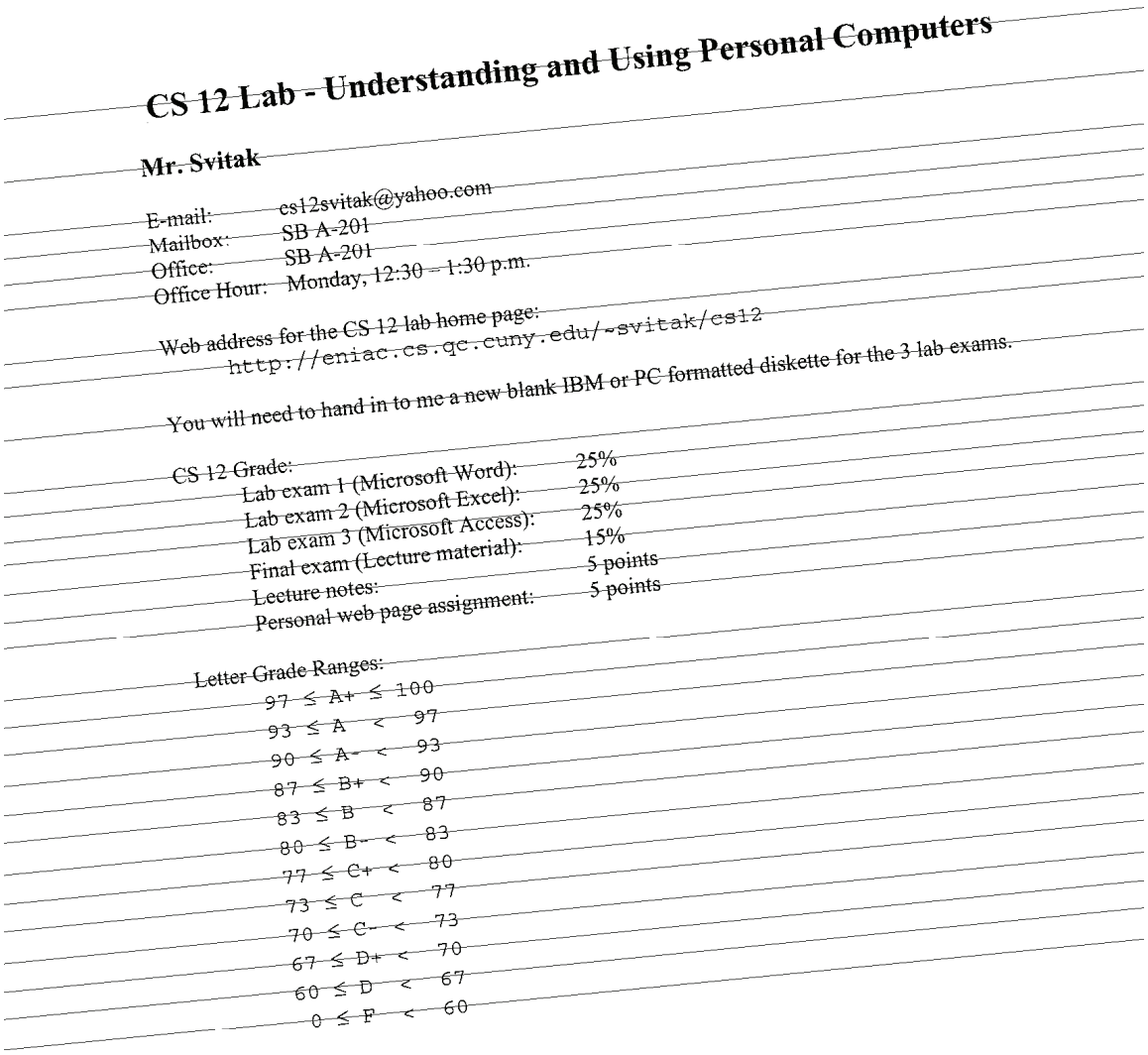
CS 12 Grade:

Lab exam 1 (Microsoft Word):	25%
Lab exam 2 (Microsoft Excel):	25%
Lab exam 3 (Microsoft Access):	25%
Final exam (Lecture material):	15%
Lecture notes:	5 points
Personal web page assignment:	5 points

Letter Grade Ranges:

```
97 ≤ A+ ≤ 100
93 ≤ A  <  97
90 ≤ A- <  93
87 ≤ B+ <  90
83 ≤ B  <  87
80 ≤ B- <  83
77 ≤ C+ <  80
73 ≤ C  <  77
70 ≤ C- <  73
67 ≤ D+ <  70
60 ≤ D  <  67
 0 ≤ F  <  60
```

Figure 6.60 - Second test page of text, not skewed.



CS 12 Lab - Understanding and Using Personal Computers

Mr. Svitak

E-mail: cs12svitak@yahoo.com
Mailbox: SB A-201
Office: SB A-201
Office Hour: Monday, 12:30 – 1:30 p.m.

Web address for the CS 12 lab home page:
`http://eniac.cs.qc.cuny.edu/~svitak/cs12`

You will need to hand in to me a new blank IBM or PC formatted diskette for the 3 lab exams.

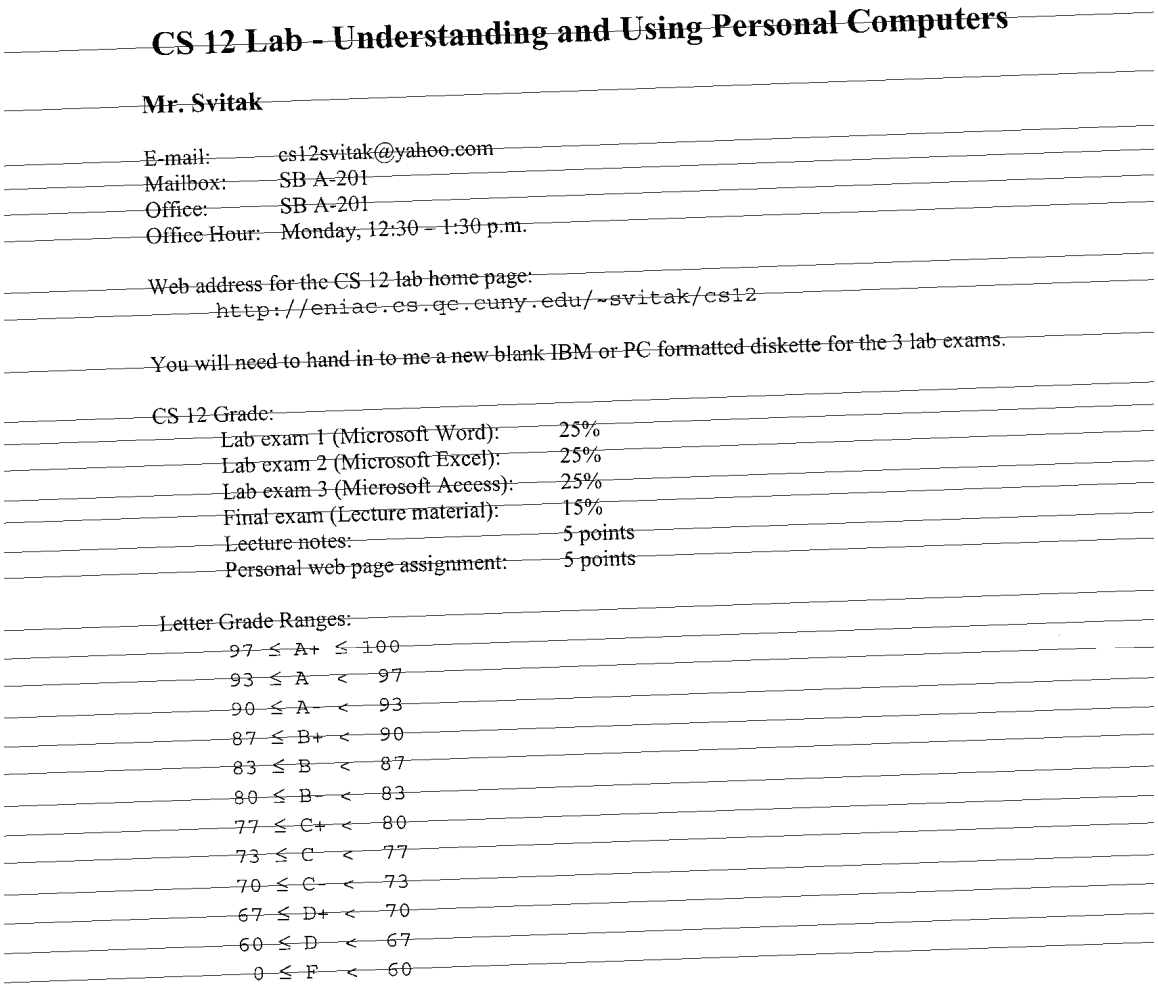
CS 12 Grade:

Lab exam 1 (Microsoft Word):	25%
Lab exam 2 (Microsoft Excel):	25%
Lab exam 3 (Microsoft Access):	25%
Final exam (Lecture material):	15%
Lecture notes:	5 points
Personal web page assignment:	5 points

Letter Grade Ranges:

```
97 ≤ A+ ≤ 100
93 ≤ A  <  97
90 ≤ A- <  93
87 ≤ B+ <  90
83 ≤ B  <  87
80 ≤ B- <  83
77 ≤ C+ <  80
73 ≤ C  <  77
70 ≤ C- <  73
67 ≤ D+ <  70
60 ≤ D  <  67
 0 ≤ F  <  60
```

Figure 6.61 - Second test page of text, skewed 6° counter-clockwise.



CS 12 Lab - Understanding and Using Personal Computers

Mr. Svitak

E-mail: cs12svitak@yahoo.com
Mailbox: SB A-201
Office: SB A-201
Office Hour: Monday, 12:30 - 1:30 p.m.

Web address for the CS 12 lab home page:
`http://eniac.cs.qc.cuny.edu/~svitak/cs12`

You will need to hand in to me a new blank IBM or PC formatted diskette for the 3 lab exams.

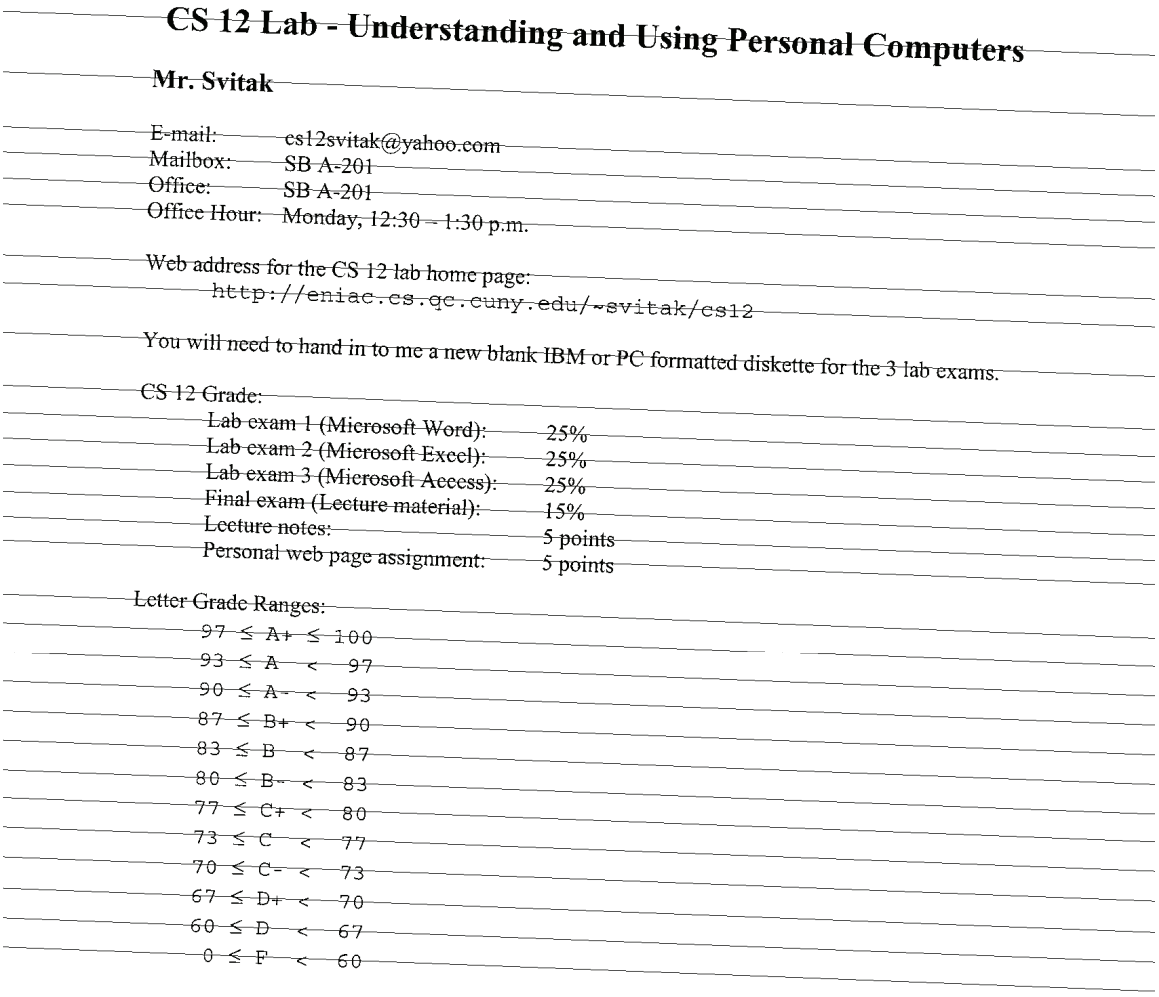
CS 12 Grade:

Lab exam 1 (Microsoft Word):	25%
Lab exam 2 (Microsoft Excel):	25%
Lab exam 3 (Microsoft Access):	25%
Final exam (Lecture material):	15%
Lecture notes:	5 points
Personal web page assignment:	5 points

Letter Grade Ranges:

```
97 ≤ A+ ≤ 100
93 ≤ A  <  97
90 ≤ A- <  93
87 ≤ B+ <  90
83 ≤ B  <  87
80 ≤ B- <  83
77 ≤ C+ <  80
73 ≤ C  <  77
70 ≤ C- <  73
67 ≤ D+ <  70
60 ≤ D  <  67
 0 ≤ F  <  60
```

Figure 6.62 - Second test page of text, skewed 2° counter-clockwise.



CS 12 Lab - Understanding and Using Personal Computers

Mr. Svitak

E-mail: cs12svitak@yahoo.com
Mailbox: SB A-201
Office: SB A-201
Office Hour: Monday, 12:30 - 1:30 p.m.

Web address for the CS 12 lab home page:
http://eniac.cs.qc.cuny.edu/~svitak/cs12

You will need to hand in to me a new blank IBM or PC formatted diskette for the 3 lab exams.

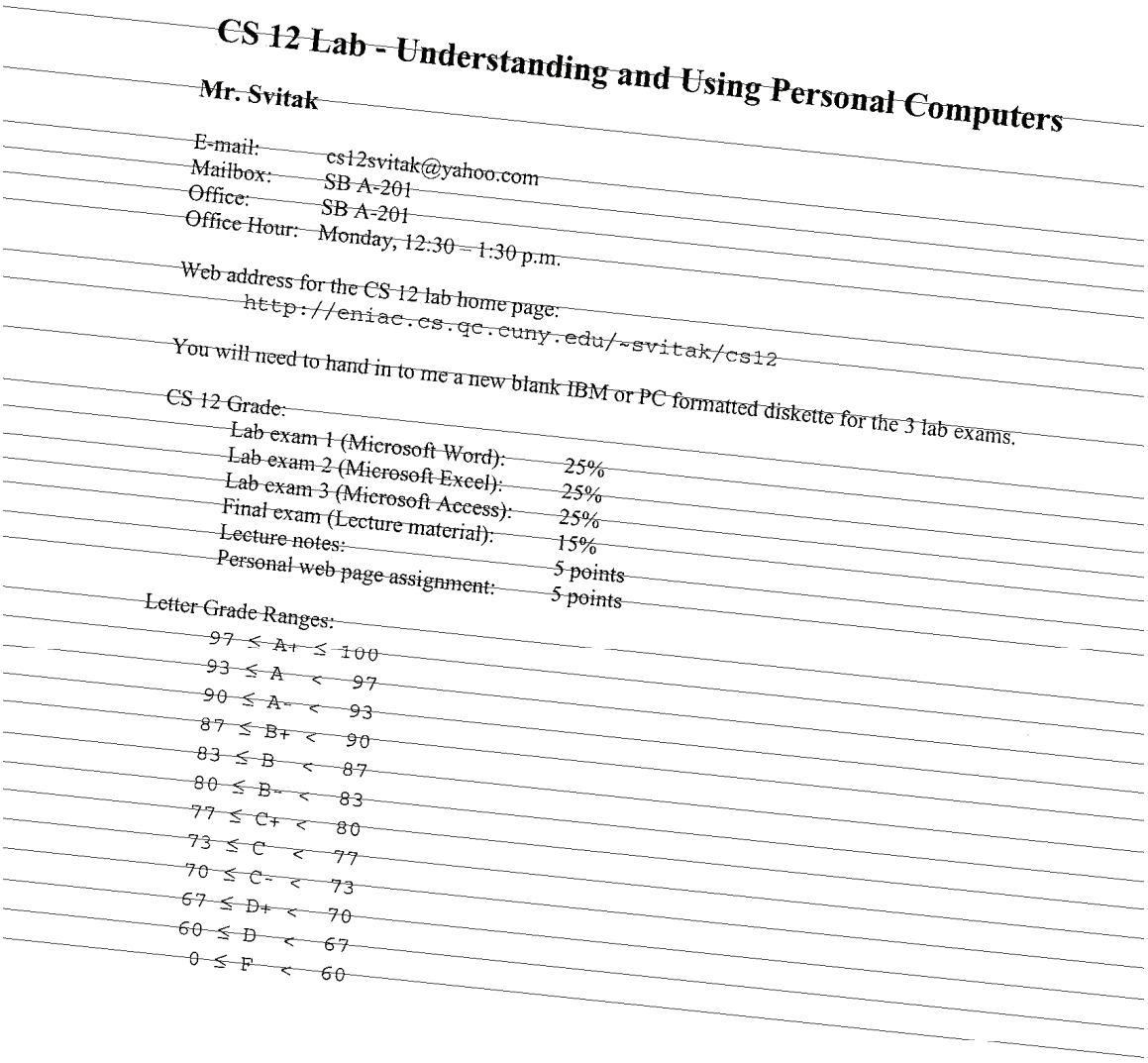
CS 12 Grade:

Lab exam 1 (Microsoft Word):	25%
Lab exam 2 (Microsoft Excel):	25%
Lab exam 3 (Microsoft Access):	25%
Final exam (Lecture material):	15%
Lecture notes:	5 points
Personal web page assignment:	5 points

Letter Grade Ranges:

97 ≤ A+ ≤ 100
93 ≤ A < 97
90 ≤ A- < 93
87 ≤ B+ < 90
83 ≤ B < 87
80 ≤ B- < 83
77 ≤ C+ < 80
73 ≤ C < 77
70 ≤ C- < 73
67 ≤ D+ < 70
60 ≤ D < 67
0 ≤ F < 60

Figure 6.63 - Second test page of text, skewed 2° clockwise.



CS 12 Lab - Understanding and Using Personal Computers

Mr. Svitak

E-mail: cs12svitak@yahoo.com
Mailbox: SB A-201
Office: SB A-201
Office Hour: Monday, 12:30 – 1:30 p.m.

Web address for the CS 12 lab home page:
`http://eniac.cs.qc.cuny.edu/~svitak/cs12`

You will need to hand in to me a new blank IBM or PC formatted diskette for the 3 lab exams.

CS 12 Grade:

Lab exam 1 (Microsoft Word):	25%
Lab exam 2 (Microsoft Excel):	25%
Lab exam 3 (Microsoft Access):	25%
Final exam (Lecture material):	15%
Lecture notes:	5 points
Personal web page assignment:	5 points

Letter Grade Ranges:

```
97 ≤ A+ ≤ 100
93 ≤ A  <  97
90 ≤ A- <  93
87 ≤ B+ <  90
83 ≤ B  <  87
80 ≤ B- <  83
77 ≤ C+ <  80
73 ≤ C  <  77
70 ≤ C- <  73
67 ≤ D+ <  70
60 ≤ D  <  67
 0 ≤ F  <  60
```

Figure 6.64 - Second test page of text, skewed 6° clockwise.

Microsoft Excel Practice Exam

1. Go to the **CSCI 012 Lab Home Page**.
2. Click on the **Practice Exam Downloads** link on the menu.
3. Follow **Instructions for Obtaining the Practice Exam File** to download and unzip the file needed to perform the practice exam.

Open the file **PracticeExcel.xls** on your storage media and save it as **MyExcel.xls**.
Your storage media is a floppy disk, a flash drive, your computer's hard drive, or your P drive.

Salary:

1. Center and merge "Expert Software Company" across cells A1 through F1. Change its font size to 16 and font color to dark blue. Change the background color of the cell containing "Expert Software Company" to yellow and change the height of row 1 to 25.

2. Adjust column widths and use functions when necessary. Enter formulas to fill in cells according to the following:

   Rows 4 through 17:
   Commission is in Column D: The salesperson's commission is based upon their total sales. **If** the salesperson's total sales are greater than or equal to $50,000 then the commission is the total sales multiplied by the commission percentage of 4%, otherwise (total sales is less than $50,000) the commission is the total sales multiplied by the commission percentage of 3%. For example, if the total sales are $35,000 then the commission is $1,050.

   See the "Commission Percentages" table in the Salary worksheet. Remember to use the percentage's cell address in your formula rather than the percentage's actual numeric value. Remember to use the commission threshold's cell address in your formula rather than the actual numeric value of 50000.

   Total Salary is in Column E: The salesperson's total salary is the base salary plus the commission.

   Salesperson's Evaluation is in Column F: Based on the salesperson's total sales, use **VLOOKUP** to assign the salesperson an evaluation remark (excellent, good, fair or poor) according to the "Evaluation Criteria" found in the Salary worksheet. For example, if the salesperson's total sales are greater than or equal to $50,000 but less than $75,000 then the evaluation remark returned by **VLOOKUP** is "Good."

   Average Total Salary is in cell E19: The average of the total salaries.

   Highest Total Salary is in cell E20: The highest total salary amongst all salespersons.

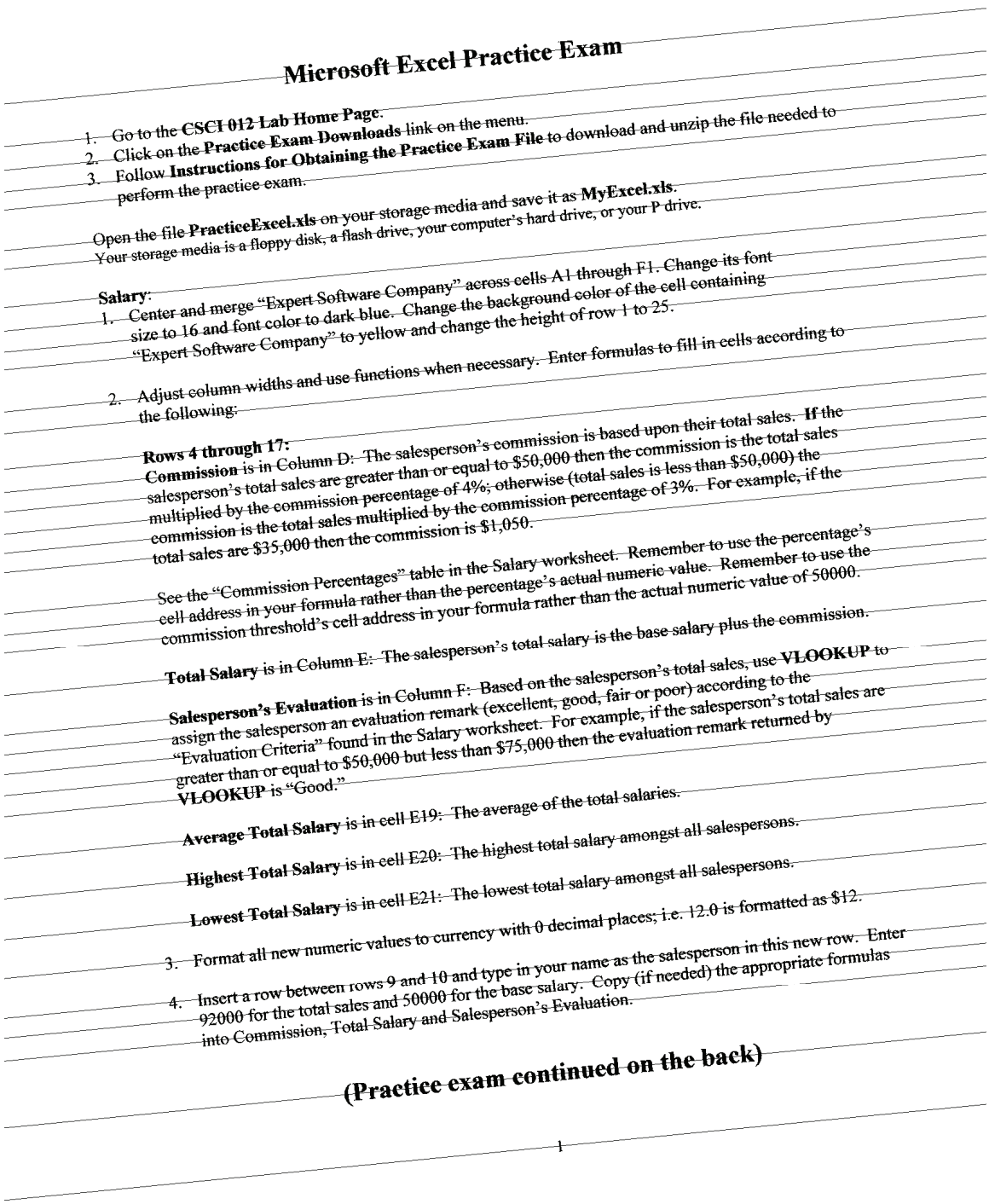
   Lowest Total Salary is in cell E21: The lowest total salary amongst all salespersons.

3. Format all new numeric values to currency with 0 decimal places; i.e. 12.0 is formatted as $12.

4. Insert a row between rows 9 and 10 and type in your name as the salesperson in this new row. Enter 92000 for the total sales and 50000 for the base salary. Copy (if needed) the appropriate formulas into Commission, Total Salary and Salesperson's Evaluation.

(Practice exam continued on the back)

Figure 6.65 - Third test page of text, not skewed.



Microsoft Excel Practice Exam

1. Go to the **CSCI 012 Lab Home Page**.
2. Click on the **Practice Exam Downloads** link on the menu.
3. Follow **Instructions for Obtaining the Practice Exam File** to download and unzip the file needed to perform the practice exam.

Open the file **PracticeExcel.xls** on your storage media and save it as **MyExcel.xls**.
Your storage media is a floppy disk, a flash drive, your computer's hard drive, or your P drive.

Salary:

1. Center and merge "Expert Software Company" across cells A1 through F1. Change its font size to 16 and font color to dark blue. Change the background color of the cell containing "Expert Software Company" to yellow and change the height of row 1 to 25.

2. Adjust column widths and use functions when necessary. Enter formulas to fill in cells according to the following:

Rows 4 through 17:
Commission is in Column D: The salesperson's commission is based upon their total sales. **If** the salesperson's total sales are greater than or equal to $50,000 then the commission is the total sales multiplied by the commission percentage of 4%; otherwise (total sales is less than $50,000) the commission is the total sales multiplied by the commission percentage of 3%. For example, if the total sales are $35,000 then the commission is $1,050.

See the "Commission Percentages" table in the Salary worksheet. Remember to use the percentage's cell address in your formula rather than the percentage's actual numeric value. Remember to use the commission threshold's cell address in your formula rather than the actual numeric value of 50000.

Total Salary is in Column E: The salesperson's total salary is the base salary plus the commission.

Salesperson's Evaluation is in Column F: Based on the salesperson's total sales, use **VLOOKUP** to assign the salesperson an evaluation remark (excellent, good, fair or poor) according to the "Evaluation Criteria" found in the Salary worksheet. For example, if the salesperson's total sales are greater than or equal to $50,000 but less than $75,000 then the evaluation remark returned by **VLOOKUP** is "Good."

Average Total Salary is in cell E19: The average of the total salaries.

Highest Total Salary is in cell E20: The highest total salary amongst all salespersons.

Lowest Total Salary is in cell E21: The lowest total salary amongst all salespersons.

3. Format all new numeric values to currency with 0 decimal places; i.e. 12.0 is formatted as $12.

4. Insert a row between rows 9 and 10 and type in your name as the salesperson in this new row. Enter 92000 for the total sales and 50000 for the base salary. Copy (if needed) the appropriate formulas into Commission, Total Salary and Salesperson's Evaluation.

(Practice exam continued on the back)

1

Figure 6.66 - Third test page of text, skewed 6° counter-clockwise.

Microsoft Excel Practice Exam

1. Go to the **CSCI 012 Lab Home Page**.
2. Click on the **Practice Exam Downloads** link on the menu.
3. Follow **Instructions for Obtaining the Practice Exam File** to download and unzip the file needed to perform the practice exam.

Open the file **PracticeExcel.xls** on your storage media and save it as **MyExcel.xls**.
Your storage media is a floppy disk, a flash drive, your computer's hard drive, or your P drive.

Salary:

1. Center and merge "Expert Software Company" across cells A1 through F1. Change its font size to 16 and font color to dark blue. Change the background color of the cell containing "Expert Software Company" to yellow and change the height of row 1 to 25.

2. Adjust column widths and use functions when necessary. Enter formulas to fill in cells according to the following.

   Rows 4 through 17:
   Commission is in Column D: The salesperson's commission is based upon their total sales. If the salesperson's total sales are greater than or equal to $50,000 then the commission is the total sales multiplied by the commission percentage of 4%; otherwise (total sales is less than $50,000) the commission is the total sales multiplied by the commission percentage of 3%. For example, if the total sales are $35,000 then the commission is $1,050.

   See the "Commission Percentages" table in the Salary worksheet. Remember to use the percentage's cell address in your formula rather than the percentage's actual numeric value. Remember to use the commission threshold's cell address in your formula rather than the actual numeric value of 50000.

   Total Salary is in Column E: The salesperson's total salary is the base salary plus the commission.

   Salesperson's Evaluation is in Column F: Based on the salesperson's total sales, use **VLOOKUP** to assign the salesperson an evaluation remark (excellent, good, fair or poor) according to the "Evaluation Criteria" found in the Salary worksheet. For example, if the salesperson's total sales are greater than or equal to $50,000 but less than $75,000 then the evaluation remark returned by **VLOOKUP** is "Good."

   Average Total Salary is in cell E19: The average of the total salaries.

   Highest Total Salary is in cell E20: The highest total salary amongst all salespersons.

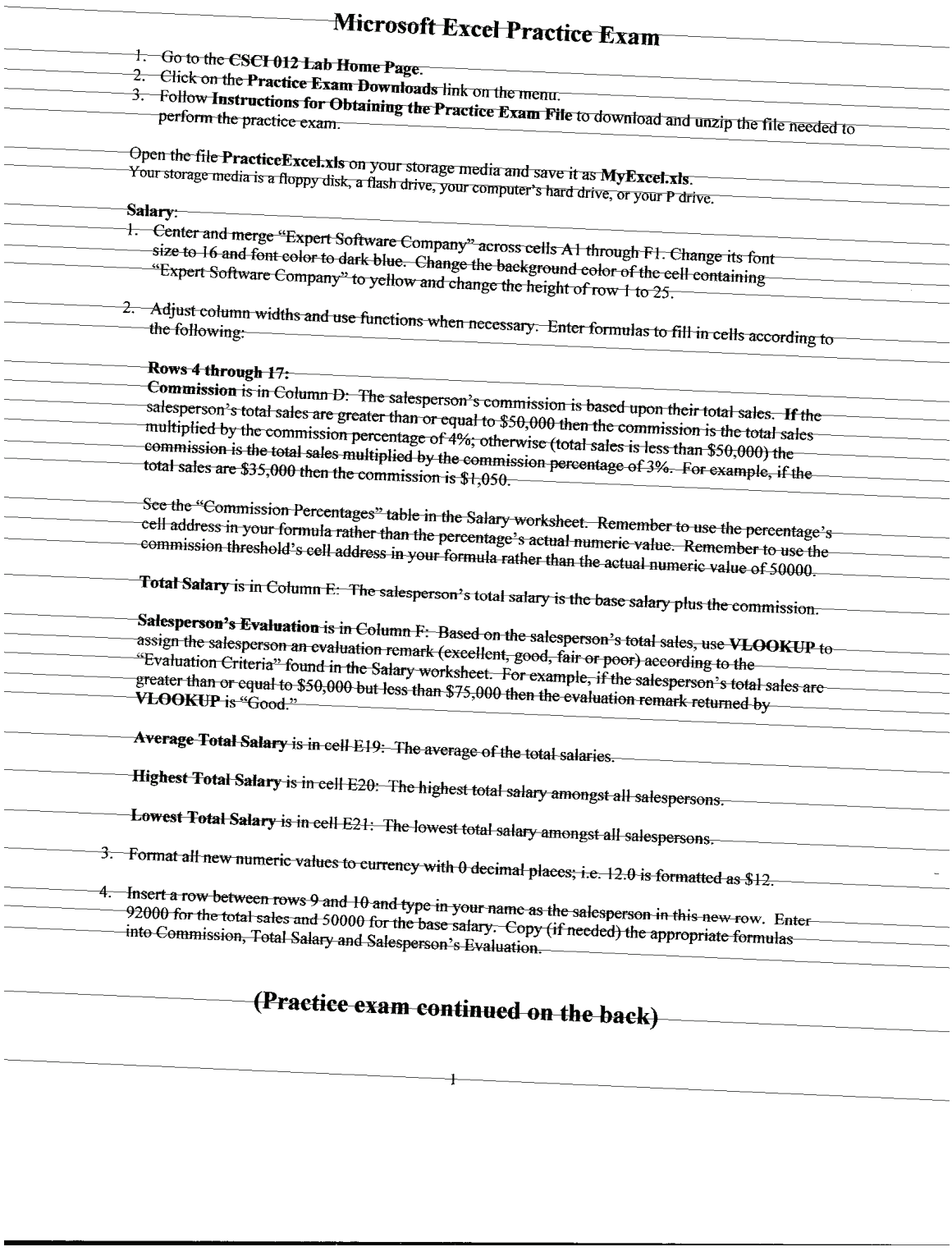
   Lowest Total Salary is in cell E21: The lowest total salary amongst all salespersons.

3. Format all new numeric values to currency with 0 decimal places; i.e. 12.0 is formatted as $12.

4. Insert a row between rows 9 and 10 and type in your name as the salesperson in this new row. Enter 92000 for the total sales and 50000 for the base salary. Copy (if needed) the appropriate formulas into Commission, Total Salary and Salesperson's Evaluation.

(Practice exam continued on the back)

Figure 6.67 - Third test page of text, skewed 2° counter-clockwise.



Microsoft Excel Practice Exam

1. Go to the **CSCI 012 Lab Home Page**.
2. Click on the **Practice Exam Downloads** link on the menu.
3. Follow **Instructions for Obtaining the Practice Exam File** to download and unzip the file needed to perform the practice exam.

Open the file **PracticeExcel.xls** on your storage media and save it as **MyExcel.xls**.
Your storage media is a floppy disk, a flash drive, your computer's hard drive, or your P drive.

Salary:

1. Center and merge "Expert Software Company" across cells A1 through F1. Change its font size to 16 and font color to dark blue. Change the background color of the cell containing "Expert Software Company" to yellow and change the height of row 1 to 25.

2. Adjust column widths and use functions when necessary. Enter formulas to fill in cells according to the following:

   Rows 4 through 17:
   Commission is in Column D: The salesperson's commission is based upon their total sales. **If** the salesperson's total sales are greater than or equal to $50,000 then the commission is the total sales multiplied by the commission percentage of 4%; otherwise (total sales is less than $50,000) the commission is the total sales multiplied by the commission percentage of 3%. For example, if the total sales are $35,000 then the commission is $1,050.

   See the "Commission Percentages" table in the Salary worksheet. Remember to use the percentage's cell address in your formula rather than the percentage's actual numeric value. Remember to use the commission threshold's cell address in your formula rather than the actual numeric value of 50000.

   Total Salary is in Column E: The salesperson's total salary is the base salary plus the commission.

   Salesperson's Evaluation is in Column F: Based on the salesperson's total sales, use **VLOOKUP** to assign the salesperson an evaluation remark (excellent, good, fair or poor) according to the "Evaluation Criteria" found in the Salary worksheet. For example, if the salesperson's total sales are greater than or equal to $50,000 but less than $75,000 then the evaluation remark returned by **VLOOKUP** is "Good."

   Average Total Salary is in cell E19: The average of the total salaries.

   Highest Total Salary is in cell E20: The highest total salary amongst all salespersons.

   Lowest Total Salary is in cell E21: The lowest total salary amongst all salespersons.

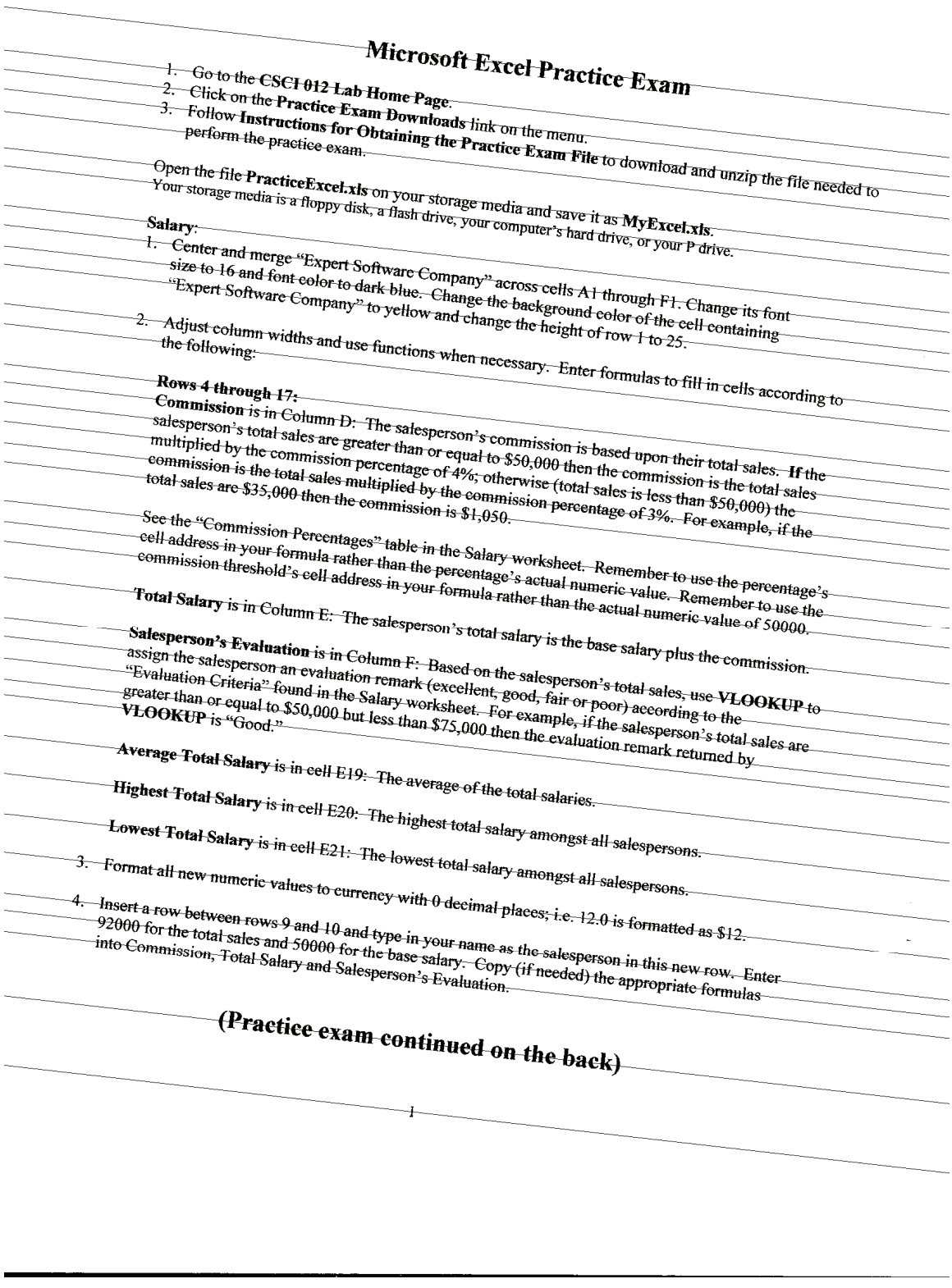
3. Format all new numeric values to currency with 0 decimal places; i.e. 12.0 is formatted as $12.

4. Insert a row between rows 9 and 10 and type in your name as the salesperson in this new row. Enter 92000 for the total sales and 50000 for the base salary. Copy (if needed) the appropriate formulas into Commission, Total Salary and Salesperson's Evaluation.

(Practice exam continued on the back)

1

Figure 6.68 - Third test page of text, skewed 2° clockwise.



Microsoft Excel Practice Exam

1. Go to the **CSCI 012 Lab Home Page**.
2. Click on the **Practice Exam Downloads** link on the menu.
3. Follow **Instructions for Obtaining the Practice Exam File** to download and unzip the file needed to perform the practice exam.

Open the file **PracticeExcel.xls** on your storage media and save it as **MyExcel.xls**.
Your storage media is a floppy disk, a flash drive, your computer's hard drive, or your P drive.

Salary:

1. Center and merge "Expert Software Company" across cells A1 through F1. Change its font size to 16 and font color to dark blue. Change the background color of the cell containing "Expert Software Company" to yellow and change the height of row 1 to 25.

2. Adjust column widths and use functions when necessary. Enter formulas to fill in cells according to the following:

Rows 4 through 17:
Commission is in Column D: The salesperson's commission is based upon their total sales. **If** the salesperson's total sales are greater than or equal to $50,000 then the commission is the total sales multiplied by the commission percentage of 4%; otherwise (total sales is less than $50,000) the commission is the total sales multiplied by the commission percentage of 3%. For example, if the total sales are $35,000 then the commission is $1,050.

See the "Commission Percentages" table in the Salary worksheet. Remember to use the percentage's cell address in your formula rather than the percentage's actual numeric value. Remember to use the commission threshold's cell address in your formula rather than the actual numeric value of 50000.

Total Salary is in Column E: The salesperson's total salary is the base salary plus the commission.

Salesperson's Evaluation is in Column F: Based on the salesperson's total sales, use **VLOOKUP** to assign the salesperson an evaluation remark (excellent, good, fair or poor) according to the "Evaluation Criteria" found in the Salary worksheet. For example, if the salesperson's total sales are greater than or equal to $50,000 but less than $75,000 then the evaluation remark returned by **VLOOKUP** is "Good."

Average Total Salary is in cell E19: The average of the total salaries.

Highest Total Salary is in cell E20: The highest total salary amongst all salespersons.

Lowest Total Salary is in cell E21: The lowest total salary amongst all salespersons.

3. Format all new numeric values to currency with 0 decimal places; i.e. 12.0 is formatted as $12.

4. Insert a row between rows 9 and 10 and type in your name as the salesperson in this new row. Enter 92000 for the total sales and 50000 for the base salary. Copy (if needed) the appropriate formulas into Commission, Total Salary and Salesperson's Evaluation.

(Practice exam continued on the back)

1

Figure 6.69 - Third test page of text, skewed 6° clockwise.

5.Press the **Save** button to save the spreadsheet.

6.Copy the entire **Salary** worksheet to Sheet2 and rename Sheet2 to **Sorted**.

Sorted:

Sort the entire list of salespersons by sorting the **Total Salaries** in descending order.

Press the **Save** button to save the spreadsheet.

Chart:

Click the worksheet tab labeled **Chart**.

Use cells A3 through E4 to form a **Pie chart with a 3-D visual effect**.

Show value for the **data labels**.

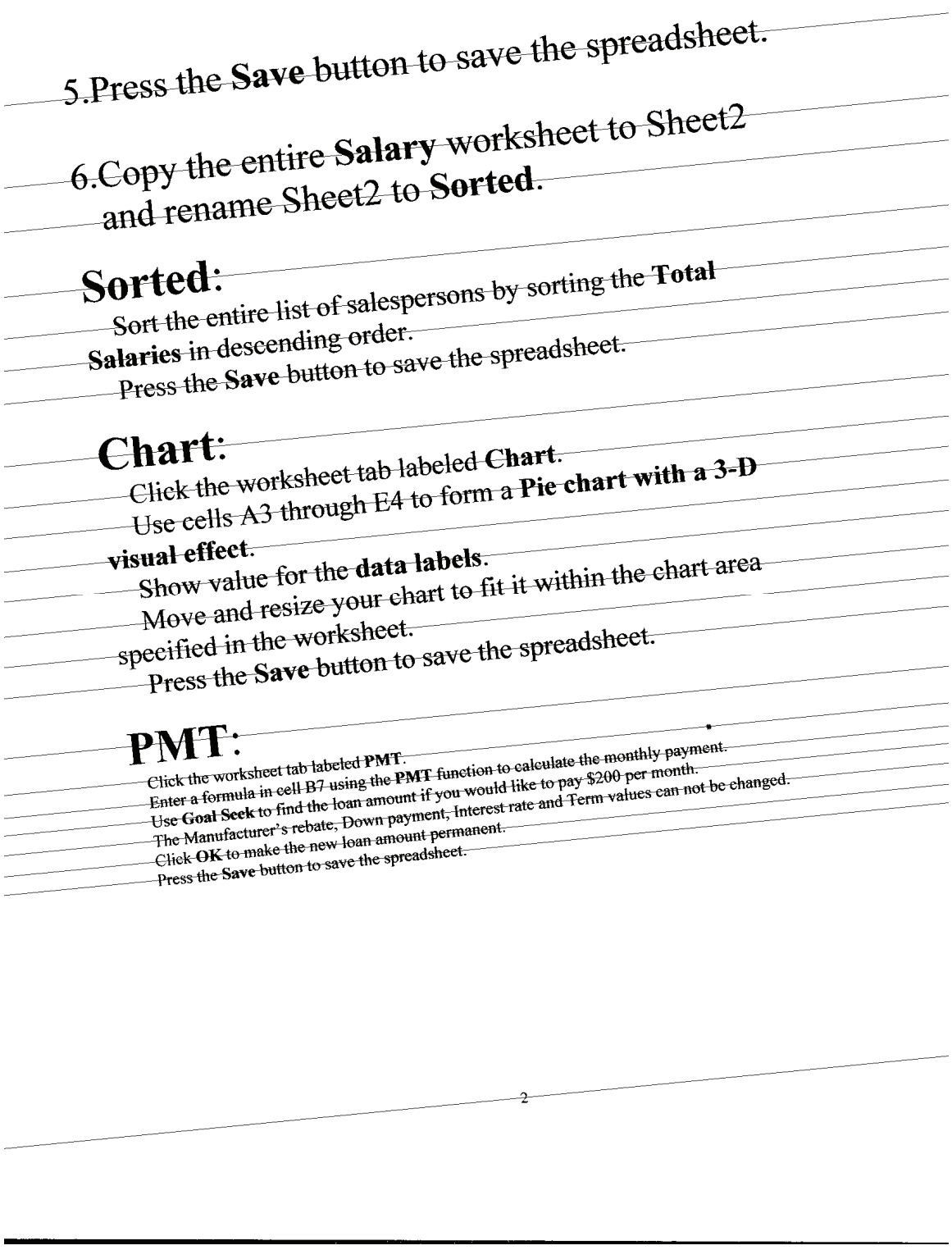
Move and resize your chart to fit it within the chart area specified in the worksheet.

Press the **Save** button to save the spreadsheet.

PMT:

Click the worksheet tab labeled **PMT**.
Enter a formula in cell B7 using the **PMT** function to calculate the monthly payment.
Use **Goal Seek** to find the loan amount if you would like to pay $200 per month.
The Manufacturer's rebate, Down payment, Interest rate and Term values can not be changed.
Click **OK** to make the new loan amount permanent.
Press the **Save** button to save the spreadsheet.

Figure 6.70 - Fourth test page of text, not skewed.



5.Press the **Save** button to save the spreadsheet.

6.Copy the entire **Salary** worksheet to Sheet2 and rename Sheet2 to **Sorted**.

Sorted:

Sort the entire list of salespersons by sorting the **Total Salaries** in descending order.
Press the **Save** button to save the spreadsheet.

Chart:

Click the worksheet tab labeled **Chart**.
Use cells A3 through E4 to form a **Pie chart with a 3-D visual effect**.
Show value for the **data labels**.
Move and resize your chart to fit it within the chart area specified in the worksheet.
Press the **Save** button to save the spreadsheet.

PMT:

Click the worksheet tab labeled **PMT**.
Enter a formula in cell B7 using the **PMT** function to calculate the monthly payment.
Use **Goal Seek** to find the loan amount if you would like to pay $200 per month.
The Manufacturer's rebate, Down payment, Interest rate and Term values can not be changed.
Click **OK** to make the new loan amount permanent.
Press the **Save** button to save the spreadsheet.

2

Figure 6.71 - Fourth test page of text, skewed 6° counter-clockwise.

5.Press the **Save** button to save the spreadsheet.

6.Copy the entire **Salary** worksheet to Sheet2 and rename Sheet2 to **Sorted**.

Sorted:

Sort the entire list of salespersons by sorting the **Total Salaries** in descending order.

Press the **Save** button to save the spreadsheet.

Chart:

Click the worksheet tab labeled **Chart**.

Use cells A3 through E4 to form a **Pie chart with a 3-D visual effect**.

Show value for the **data labels**.

Move and resize your chart to fit it within the chart area specified in the worksheet.

Press the **Save** button to save the spreadsheet.

PMT:

Click the worksheet tab labeled **PMT**.
Enter a formula in cell B7 using the **PMT** function to calculate the monthly payment.
Use **Goal Seek** to find the loan amount if you would like to pay $200 per month.
The Manufacturer's rebate, Down payment, Interest rate and Term values can not be changed.
Click **OK** to make the new loan amount permanent.
Press the **Save** button to save the spreadsheet.

Figure 6.72 - Fourth test page of text, skewed 2° counter-clockwise.

5.Press the **Save** button to save the spreadsheet.

6.Copy the entire **Salary** worksheet to Sheet2 and rename Sheet2 to **Sorted**.

Sorted:

Sort the entire list of salespersons by sorting the **Total Salaries** in descending order.
Press the **Save** button to save the spreadsheet.

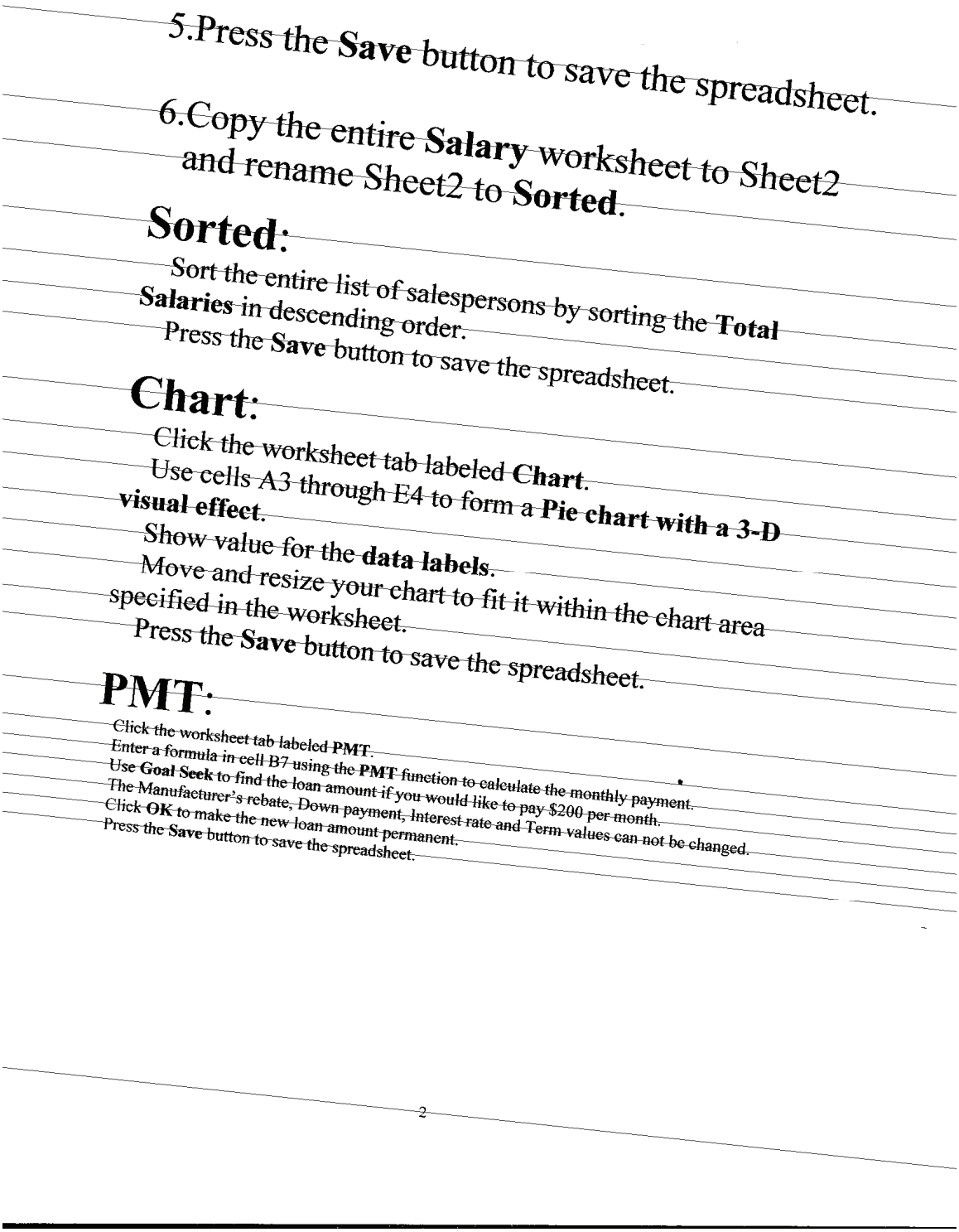
Chart:

Click the worksheet tab labeled **Chart**.
Use cells A3 through E4 to form a **Pie chart with a 3-D visual effect**.
Show value for the **data labels**.
Move and resize your chart to fit it within the chart area specified in the worksheet.
Press the **Save** button to save the spreadsheet.

PMT:

Click the worksheet tab labeled **PMT**.
Enter a formula in cell B7 using the **PMT** function to calculate the monthly payment.
Use **Goal Seek** to find the loan amount if you would like to pay $200 per month.
The Manufacturer's rebate, Down payment, Interest rate and Term values can not be changed.
Click **OK** to make the new loan amount permanent.
Press the **Save** button to save the spreadsheet.

Figure 6.73 - Fourth test page of text, skewed 2° clockwise.



5.Press the **Save** button to save the spreadsheet.

6.Copy the entire **Salary** worksheet to Sheet2 and rename Sheet2 to **Sorted**.

Sorted:

Sort the entire list of salespersons by sorting the **Total Salaries** in descending order.
Press the **Save** button to save the spreadsheet.

Chart:

Click the worksheet tab labeled **Chart**.
Use cells A3 through E4 to form a **Pie chart with a 3-D visual effect**.
Show value for the **data labels**.
Move and resize your chart to fit it within the chart area specified in the worksheet.
Press the **Save** button to save the spreadsheet.

PMT:

Click the worksheet tab labeled **PMT**.
Enter a formula in cell B7 using the **PMT** function to calculate the monthly payment.
Use **Goal Seek** to find the loan amount if you would like to pay $200 per month.
The Manufacturer's rebate, Down payment, Interest rate and Term values can not be changed.
Click **OK** to make the new loan amount permanent.
Press the **Save** button to save the spreadsheet.

2

Figure 6.74 - Fourth test page of text, skewed 6° clockwise.

ORAL HYGIENE INSTRUCTION FOLLOWING SUTURE REMOVAL

The three weeks following periodontal surgery are the most crucial in terms of post-operative healing and long-term success. While the areas operated on may be very sore and sensitive, it is of the utmost importance that the area be kept clean of bacterial plaque. You may begin to chew on the side where surgery was performed.

The following is a day-to-day instruction in post-operative oral hygiene. While the procedures may lead to bleeding and discomfort, please follow them carefully. AVOIDING THE POST-OPERATIVE SITE MAY LEAD TO FAILURE OF THE PROCEDURE.

DAY 1 & 2 Using a Q-tip moistened with warm water, swab the gum tissue and dab the spaces between the teeth a minimum of five or six times a day.

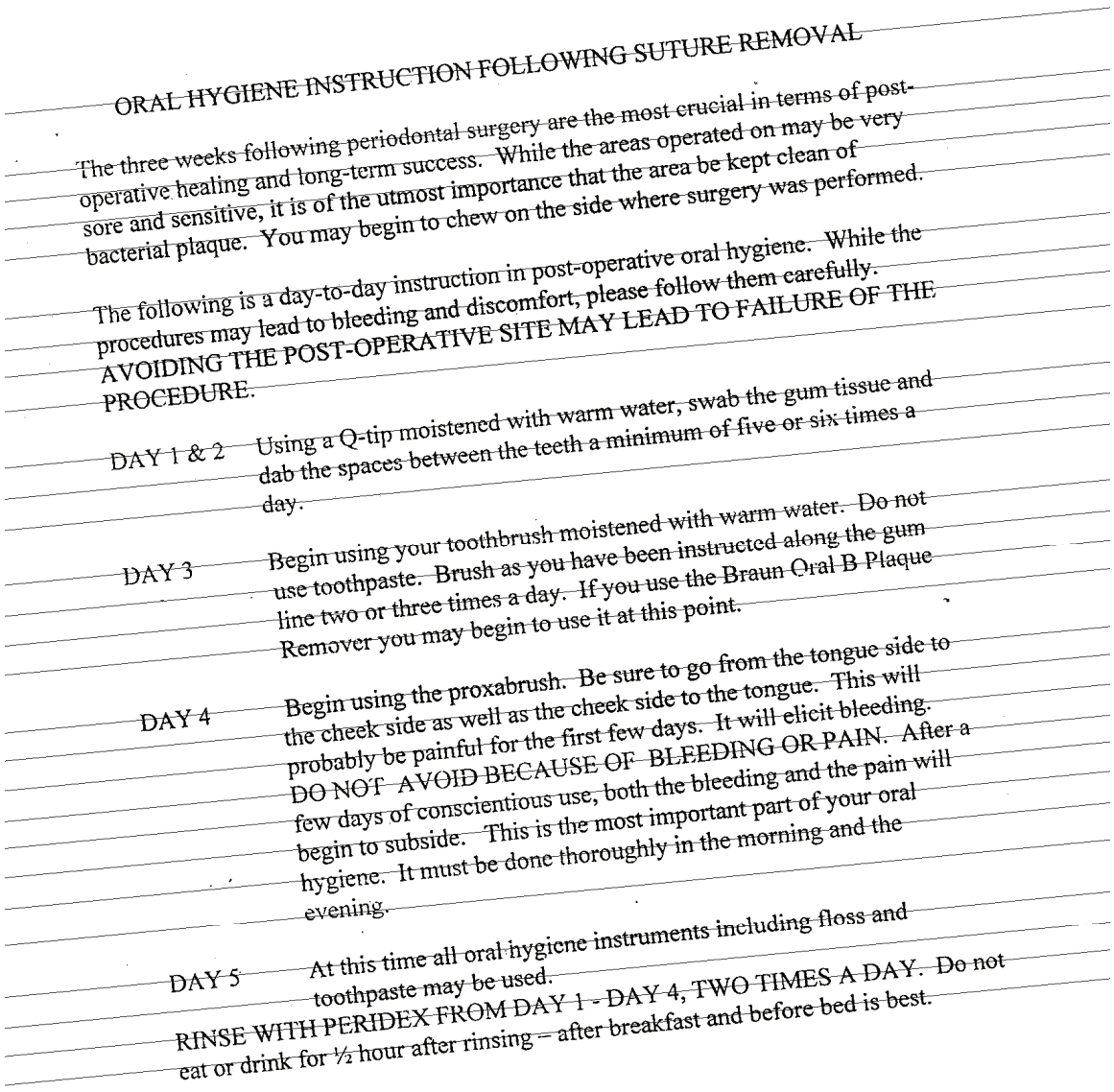
DAY 3 Begin using your toothbrush moistened with warm water. Do not use toothpaste. Brush as you have been instructed along the gum line two or three times a day. If you use the Braun Oral B Plaque Remover you may begin to use it at this point.

DAY 4 Begin using the proxabrush. Be sure to go from the tongue side to the cheek side as well as the cheek side to the tongue. This will probably be painful for the first few days. It will elicit bleeding. DO NOT AVOID BECAUSE OF BLEEDING OR PAIN. After a few days of conscientious use, both the bleeding and the pain will begin to subside. This is the most important part of your oral hygiene. It must be done thoroughly in the morning and the evening.

DAY 5 At this time all oral hygiene instruments including floss and toothpaste may be used.

RINSE WITH PERIDEX FROM DAY 1 - DAY 4, TWO TIMES A DAY. Do not eat or drink for ½ hour after rinsing – after breakfast and before bed is best.

Figure 6.75 - Fifth test page of text, not skewed.



ORAL HYGIENE INSTRUCTION FOLLOWING SUTURE REMOVAL

The three weeks following periodontal surgery are the most crucial in terms of post-operative healing and long-term success. While the areas operated on may be very sore and sensitive, it is of the utmost importance that the area be kept clean of bacterial plaque. You may begin to chew on the side where surgery was performed.

The following is a day-to-day instruction in post-operative oral hygiene. While the procedures may lead to bleeding and discomfort, please follow them carefully. AVOIDING THE POST-OPERATIVE SITE MAY LEAD TO FAILURE OF THE PROCEDURE.

DAY 1 & 2 Using a Q-tip moistened with warm water, swab the gum tissue and dab the spaces between the teeth a minimum of five or six times a day.

DAY 3 Begin using your toothbrush moistened with warm water. Do not use toothpaste. Brush as you have been instructed along the gum line two or three times a day. If you use the Braun Oral B Plaque Remover you may begin to use it at this point.

DAY 4 Begin using the proxabrush. Be sure to go from the tongue side to the cheek side as well as the cheek side to the tongue. This will probably be painful for the first few days. It will elicit bleeding. DO NOT AVOID BECAUSE OF BLEEDING OR PAIN. After a few days of conscientious use, both the bleeding and the pain will begin to subside. This is the most important part of your oral hygiene. It must be done thoroughly in the morning and the evening.

DAY 5 At this time all oral hygiene instruments including floss and toothpaste may be used.

RINSE WITH PERIDEX FROM DAY 1 - DAY 4, TWO TIMES A DAY. Do not eat or drink for ½ hour after rinsing – after breakfast and before bed is best.

Figure 6.76 - Fifth test page of text, skewed 6° counter-clockwise.

ORAL HYGIENE INSTRUCTION FOLLOWING SUTURE REMOVAL

The three weeks following periodontal surgery are the most crucial in terms of post-operative healing and long-term success. While the areas operated on may be very sore and sensitive, it is of the utmost importance that the area be kept clean of bacterial plaque. You may begin to chew on the side where surgery was performed.

The following is a day-to-day instruction in post-operative oral hygiene. While the procedures may lead to bleeding and discomfort, please follow them carefully. AVOIDING THE POST-OPERATIVE SITE MAY LEAD TO FAILURE OF THE PROCEDURE.

DAY 1 & 2 Using a Q-tip moistened with warm water, swab the gum tissue and dab the spaces between the teeth a minimum of five or six times a day.

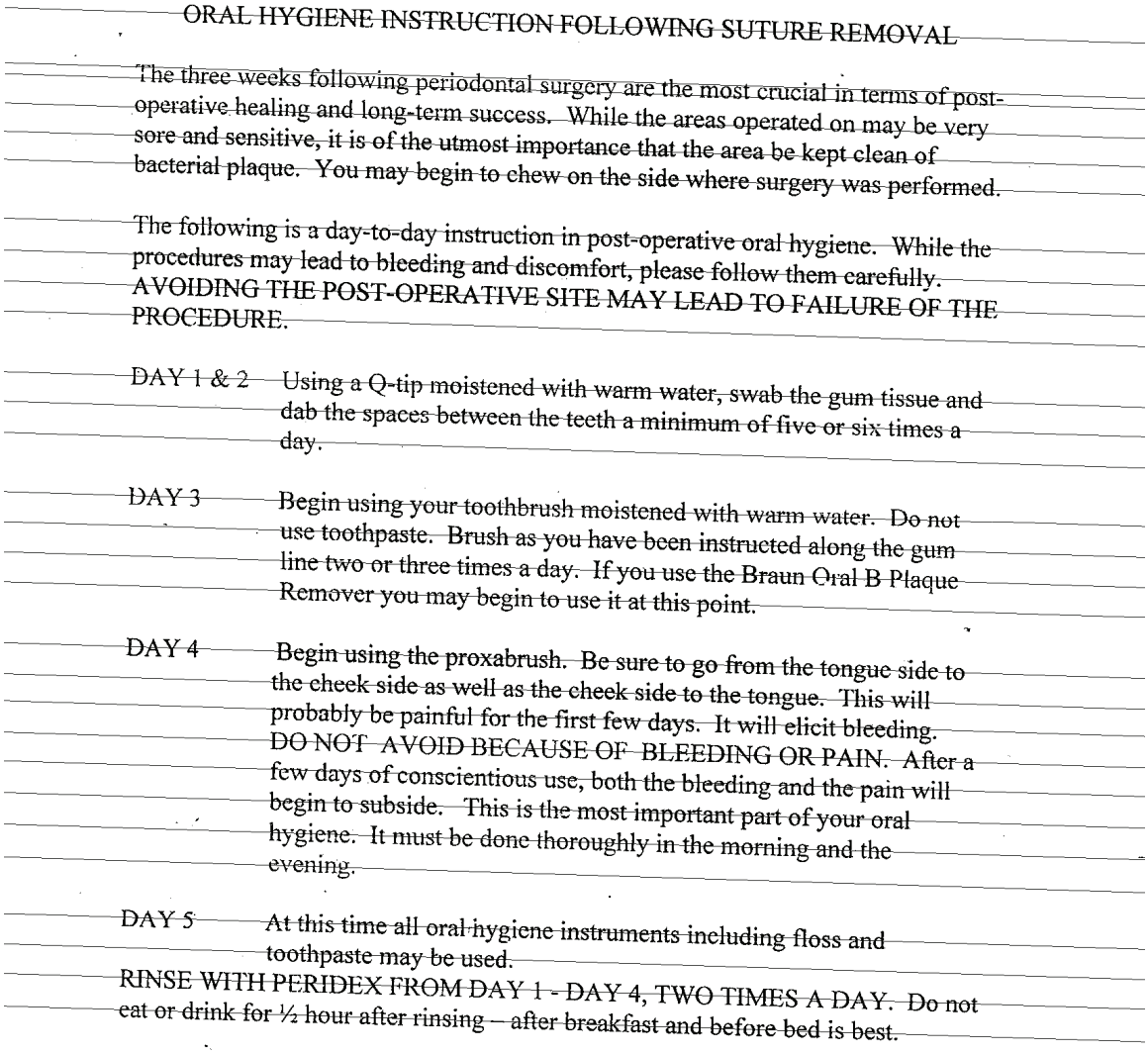
DAY 3 Begin using your toothbrush moistened with warm water. Do not use toothpaste. Brush as you have been instructed along the gum line two or three times a day. If you use the Braun Oral B Plaque Remover you may begin to use it at this point.

DAY 4 Begin using the proxabrush. Be sure to go from the tongue side to the cheek side as well as the cheek side to the tongue. This will probably be painful for the first few days. It will elicit bleeding. DO NOT AVOID BECAUSE OF BLEEDING OR PAIN. After a few days of conscientious use, both the bleeding and the pain will begin to subside. This is the most important part of your oral hygiene. It must be done thoroughly in the morning and the evening.

DAY 5 At this time all oral hygiene instruments including floss and toothpaste may be used.

RINSE WITH PERIDEX FROM DAY 1 - DAY 4, TWO TIMES A DAY. Do not eat or drink for ½ hour after rinsing – after breakfast and before bed is best.

Figure 6.77 - Fifth test page of text, skewed 2° counter-clockwise.



ORAL HYGIENE INSTRUCTION FOLLOWING SUTURE REMOVAL

The three weeks following periodontal surgery are the most crucial in terms of post-operative healing and long-term success. While the areas operated on may be very sore and sensitive, it is of the utmost importance that the area be kept clean of bacterial plaque. You may begin to chew on the side where surgery was performed.

The following is a day-to-day instruction in post-operative oral hygiene. While the procedures may lead to bleeding and discomfort, please follow them carefully. AVOIDING THE POST-OPERATIVE SITE MAY LEAD TO FAILURE OF THE PROCEDURE.

DAY 1 & 2 Using a Q-tip moistened with warm water, swab the gum tissue and dab the spaces between the teeth a minimum of five or six times a day.

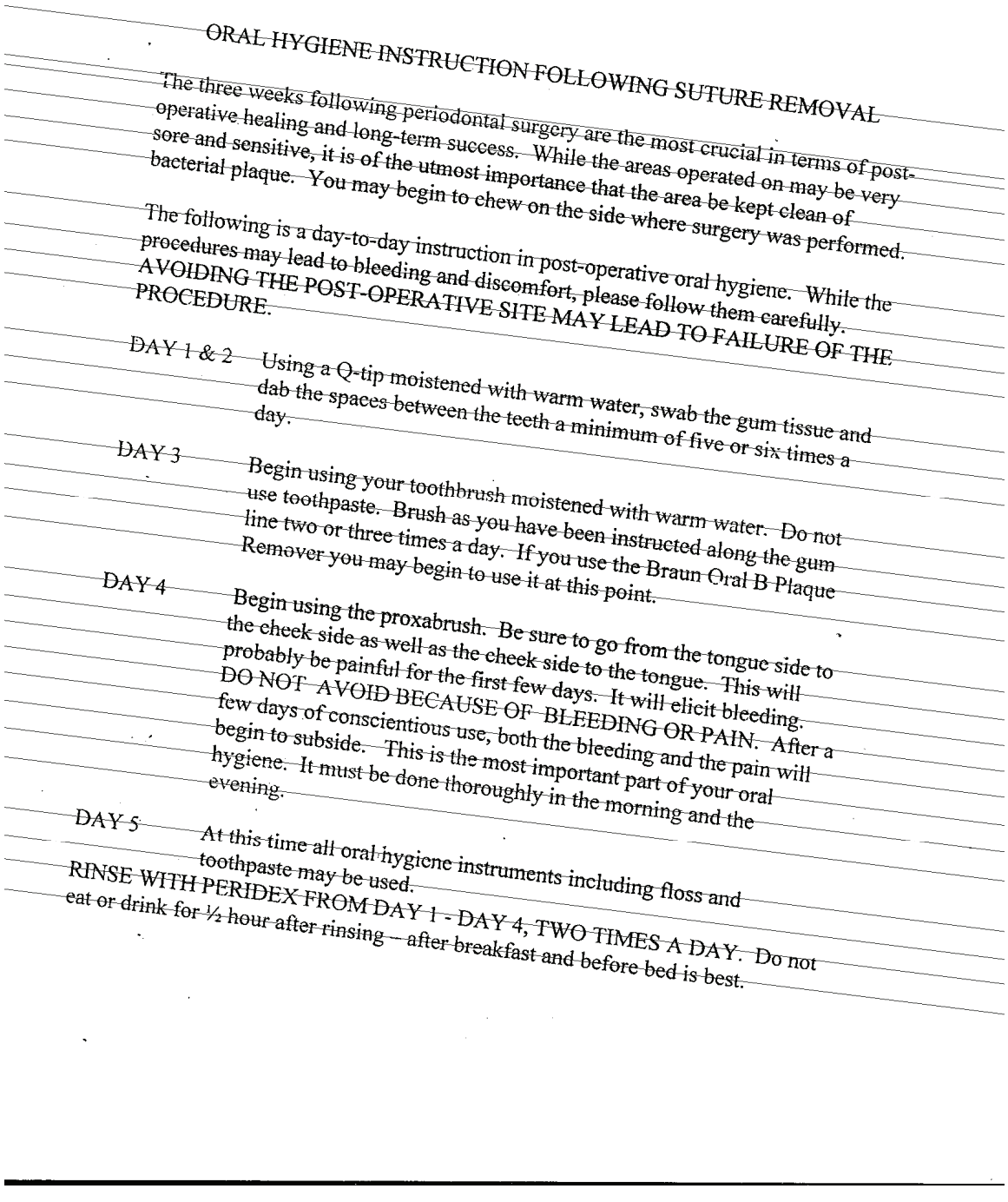
DAY 3 Begin using your toothbrush moistened with warm water. Do not use toothpaste. Brush as you have been instructed along the gum line two or three times a day. If you use the Braun Oral B Plaque Remover you may begin to use it at this point.

DAY 4 Begin using the proxabrush. Be sure to go from the tongue side to the cheek side as well as the cheek side to the tongue. This will probably be painful for the first few days. It will elicit bleeding. DO NOT AVOID BECAUSE OF BLEEDING OR PAIN. After a few days of conscientious use, both the bleeding and the pain will begin to subside. This is the most important part of your oral hygiene. It must be done thoroughly in the morning and the evening.

DAY 5 At this time all oral hygiene instruments including floss and toothpaste may be used.

RINSE WITH PERIDEX FROM DAY 1 - DAY 4, TWO TIMES A DAY. Do not eat or drink for ½ hour after rinsing – after breakfast and before bed is best.

Figure 6.78 - Fifth test page of text, skewed 2° clockwise.



ORAL HYGIENE INSTRUCTION FOLLOWING SUTURE REMOVAL

The three weeks following periodontal surgery are the most crucial in terms of post-operative healing and long-term success. While the areas operated on may be very sore and sensitive, it is of the utmost importance that the area be kept clean of bacterial plaque. You may begin to chew on the side where surgery was performed.

The following is a day-to-day instruction in post-operative oral hygiene. While the procedures may lead to bleeding and discomfort, please follow them carefully. AVOIDING THE POST-OPERATIVE SITE MAY LEAD TO FAILURE OF THE PROCEDURE.

DAY 1 & 2 Using a Q-tip moistened with warm water, swab the gum tissue and dab the spaces between the teeth a minimum of five or six times a day.

DAY 3 Begin using your toothbrush moistened with warm water. Do not use toothpaste. Brush as you have been instructed along the gum line two or three times a day. If you use the Braun Oral B Plaque Remover you may begin to use it at this point.

DAY 4 Begin using the proxabrush. Be sure to go from the tongue side to the cheek side as well as the cheek side to the tongue. This will probably be painful for the first few days. It will elicit bleeding. DO NOT AVOID BECAUSE OF BLEEDING OR PAIN. After a few days of conscientious use, both the bleeding and the pain will begin to subside. This is the most important part of your oral hygiene. It must be done thoroughly in the morning and the evening.

DAY 5 At this time all oral hygiene instruments including floss and toothpaste may be used.

RINSE WITH PERIDEX FROM DAY 1 - DAY 4, TWO TIMES A DAY. Do not eat or drink for ½ hour after rinsing – after breakfast and before bed is best.

Figure 6.79 - Fifth test page of text, skewed 6° clockwise.

Figure 6.80 - Sixth test page of text, not skewed.

Setting Up Your Web Page

A description of the web page assignment can be found on the CSCI 012 Lab web site. There is more to the web page assignment than what you do in lab class today, as you can see from the web page assignment. There are addition resources to help you complete your web page assignment; the **UNIX** and **Introduction to the Internet** sections in your **Lecture Notes** book, Dr. Waxman's own lecture material on the subject and my tutorial on the subject found on the CSCI 012 Lab web site.

In lab today, you will create and setup the necessary directory and main web page so you will be able to successfully complete your web page assignment. The web page assignment is always done in your UNIX account. Complete part A, followed by part B and finally followed by part C. In part C, complete section 1 followed by section 2. Please read everything in this handout. Especially, the text after the commands you are asked to execute because the text gives an explanation of why you are executing the command.

A. Logon to your UNIX account and wait for the operating system prompt, **$**, to appear. You log into your UNIX account each time you want to work on your web pages for the web page assignment.

B. The following three commands you will do **once** and only **once**. These commands set up your UNIX account so that you are able to view your web pages on the Internet.

1. Type in the following command at the operating system prompt and press the enter key.

```
chmod 755 ~
```

This command sets the proper permissions for the home directory of your UNIX account.

2. Type in the following command at the operating system prompt and press the enter key.

```
mkdir public_html
```

This command creates a directory called **public_html** in the home directory of your UNIX account.

3. Type in the following command at the operating system prompt and press the enter key.

1

Figure 6.81 - Sixth test page of text, skewed 6° counter-clockwise.

Setting Up Your Web Page

A description of the web page assignment can be found on the CSCI 012 Lab web site. There is more to the web page assignment then what you do in lab class today, as you can see from the web page assignment. There are addition resources to help you complete your web page assignment; the **UNIX** and **Introduction to the Internet** sections in your **Lecture Notes** book, Dr. Waxman's own lecture material on the subject and my tutorial on the subject found on the CSCI 012 Lab web site.

In lab today, you will create and setup the necessary directory and main web page so you will be able to successfully complete your web page assignment. The web page assignment is always done in your UNIX account. Complete part A, followed by part B and finally followed by part C. In part C, complete section 1 followed by section 2. Please read everything in this handout. Especially, the text after the commands you are asked to execute because the text gives an explanation of why you are executing the command.

A. Logon to your UNIX account and wait for the operating system prompt, **$**, to appear. You log into your UNIX account each time you want to work on your web pages for the web page assignment.

B. The following three commands you will do **once** and only **once**. These commands set up your UNIX account so that you are able to view your web pages on the Internet.

1. Type in the following command at the operating system prompt and press the enter key.

```
chmod 755 ~
```

This command sets the proper permissions for the home directory of your UNIX account.

2. Type in the following command at the operating system prompt and press the enter key.

```
mkdir public_html
```

This command creates a directory called **public_html** in the home directory of your UNIX account.

3. Type in the following command at the operating system prompt and press the enter key.

1

Figure 6.82 - Sixth test page of text, skewed 2° counter-clockwise.

Setting Up Your Web Page

A description of the web page assignment can be found on the CSCI 012 Lab web site. There is more to the web page assignment then what you do in lab class today, as you can see from the web page assignment. There are addition resources to help you complete your web page assignment; the **UNIX** and **Introduction to the Internet** sections in your **Lecture Notes** book, Dr. Watman's own lecture material on the subject and my tutorial on the subject found on the CSCI 012 Lab web site.

In lab today, you will create and setup the necessary directory and main web page so you will be able to successfully complete your web page assignment. The web page assignment is always done in your UNIX account. Complete part A, followed by part B and finally followed by part C. In part C, complete section 1 followed by section 2. Please read everything in this handout. Especially, the text after the commands you are asked to execute because the text gives an explanation of why you are executing the command.

A. Logon to your UNIX account and wait for the operating system prompt, **$**, to appear. You log into your UNIX account each time you want to work on your web pages for the web page assignment.

B. The following three commands you will do **once** and only **once**. These commands set up your UNIX account so that you are able to view your web pages on the Internet.

1. Type in the following command at the operating system prompt and press the enter key.

```
chmod 755 ~
```

This command sets the proper permissions for the home directory of your UNIX account.

2. Type in the following command at the operating system prompt and press the enter key.

```
mkdir public_html
```

This command creates a directory called **public_html** in the home directory of your UNIX account.

3. Type in the following command at the operating system prompt and press the enter key.

Figure 6.83 - Sixth test page of text, skewed 2° clockwise.

Setting Up Your Web Page

A description of the web page assignment can be found on the CSCI 012 Lab web site. There is more to the web page assignment than what you do in lab class today, as you can see from the web page assignment. There are additional resources to help you complete your web page assignment; the **UNIX** and **Introduction to the Internet** sections in your **Lecture Notes** book, Dr. Waxman's own lecture material on the subject and my tutorial on the subject found on the CSCI 012 Lab web site.

In lab today, you will create and setup the necessary directory and main web page so you will be able to successfully complete your web page assignment. The web page assignment is always done in your UNIX account. Complete part A, followed by part B and finally followed by part C. In part C, complete section 1 followed by section 2. Please read everything in this handout. Especially the text after the commands you are asked to execute because the text gives an explanation of why you are executing the command.

A. Logon to your UNIX account and wait for the operating system prompt, **$**, to appear. You log into your UNIX account each time you want to work on your web pages for the web page assignment.

B. The following three commands you will do **once** and only **once**. These commands set up your UNIX account so that you are able to view your web pages on the Internet.

1. Type in the following command at the operating system prompt and press the enter key.

```
chmod 755 ~
```

This command sets the proper permissions for the home directory of your UNIX account.

2. Type in the following command at the operating system prompt and press the enter key.

```
mkdir public_html
```

This command creates a directory called **public_html** in the home directory of your UNIX account.

3. Type in the following command at the operating system prompt and press the enter key.

1

Figure 6.84 - Sixth test page of text, skewed 6° clockwise.

1. Create a Report:

   Use the Report Wizard and choose the **Books** table to **create** a report having the following requirements:

   - Select the **Author**, **Year**, **ListPrice** and **Publisher** fields.
   - Do not choose any grouping levels.
   - Sort the report by 1) the **Year** field in **descending** order and 2) the **ListPrice** field in **ascending** order.
   - Choose **Tabular** layout and **Portrait** orientation.
   - Choose the **Bold** style.
   - Type **Book List** as the report's title.

2. Create Queries:

   Query 1: List all of the books that are published by **Prentice Hall** using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the **ListPrice** field. Save this query as **Prentice Hall Books**.

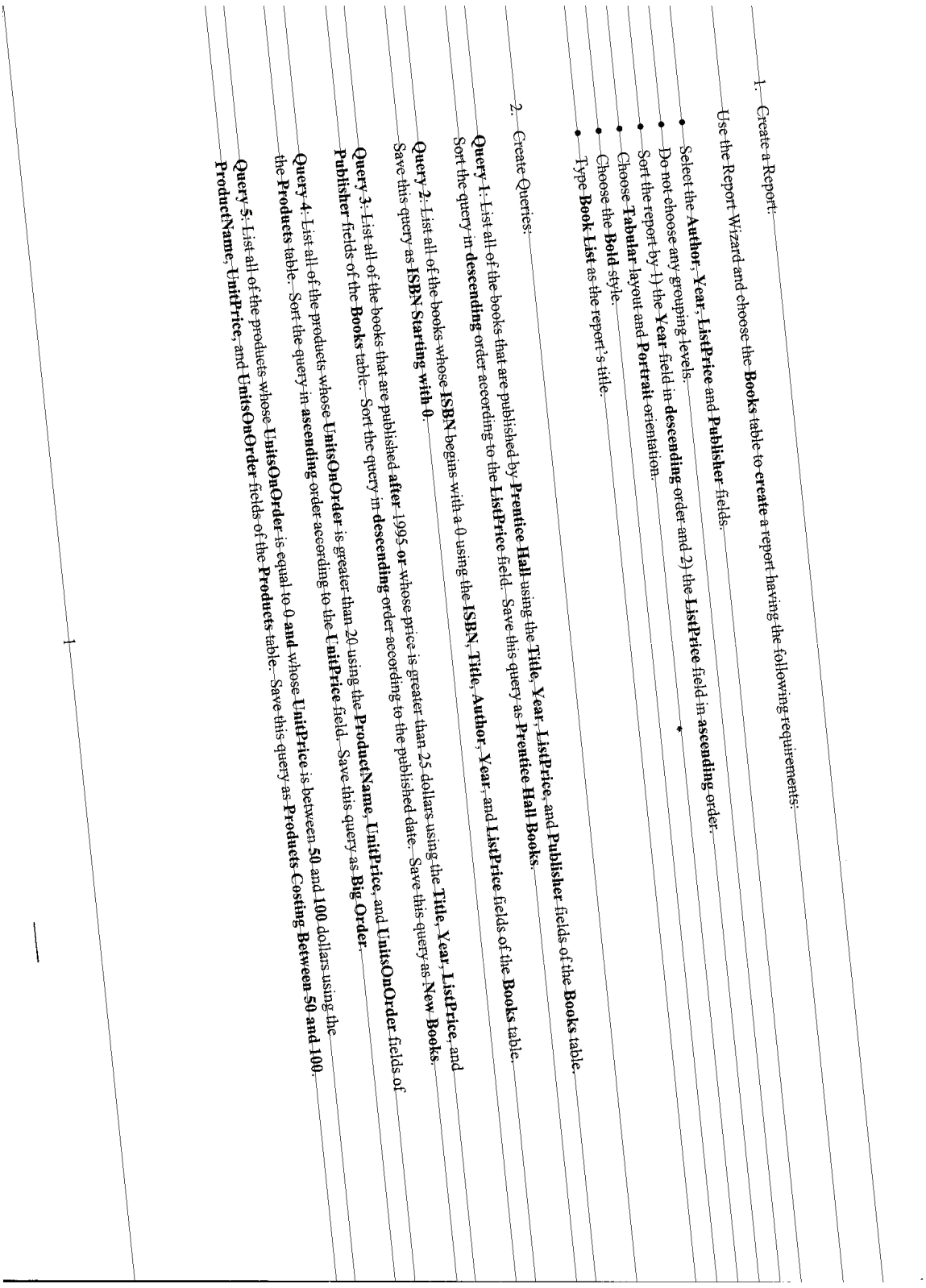
   Query 2: List all of the books whose **ISBN** begins with a 0 using the **ISBN**, **Title**, **Author**, **Year**, and **ListPrice** fields of the **Books** table. Save this query as **ISBN Starting with 0**.

   Query 3: List all of the books that are published **after** 1995 **or** whose price is greater than 25 dollars using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the published date. Save this query as **New Books**.

   Query 4: List all of the products whose **UnitsOnOrder** is greater than 20 using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Sort the query in **ascending** order according to the **UnitPrice** field. Save this query as **Big Order**.

   Query 5: List all of the products whose **UnitsOnOrder** is equal to 0 **and** whose **UnitPrice** is between **50** and **100** dollars using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Save this query as **Products Costing Between 50 and 100**.

Figure 6.85 - Seventh test page of text, not skewed.



1. Create a Report:

Use the Report Wizard and choose the **Books** table to **create** a report having the following requirements:

- Select the **Author**, **Year**, **ListPrice** and **Publisher** fields.
- Do not choose any grouping levels.
- Sort the report by 1) the **Year** field in **descending** order and 2) the **ListPrice** field in **ascending** order.
- Choose **Tabular** layout and **Portrait** orientation.
- Choose the **Bold** style.
- Type **Book List** as the report's title.

2. Create Queries:

Query 1: List all of the books that are published by **Prentice Hall** using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the **ListPrice** field. Save this query as **Prentice Hall Books**.

Query 2: List all of the books whose **ISBN** begins with a 0 using the **ISBN**, **Title**, **Author**, **Year**, and **ListPrice** fields of the **Books** table. Save this query as **ISBN Starting with 0**.

Query 3: List all of the books that are published **after** 1995 **or** whose price is greater than 25 dollars using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the published date. Save this query as **New Books**.

Query 4: List all of the products whose **UnitsOnOrder** is greater than 20 using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Sort the query in **ascending** order according to the **UnitPrice** field. Save this query as **Big Order**.

Query 5: List all of the products whose **UnitsOnOrder** is equal to 0 **and** whose **UnitPrice** is between **50** and **100** dollars using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Save this query as **Products Costing Between 50 and 100**.

Figure 6.86 - Seventh test page of text, skewed 6° counter-clockwise.

1. Create a Report:

Use the Report Wizard and choose the **Books** table to **create** a report having the following requirements:

- Select the **Author**, **Year**, **ListPrice** and **Publisher** fields.
- Do not choose any grouping levels.
- Sort the report by 1) the **Year** field in **descending** order and 2) the **ListPrice** field in **ascending** order.
- Choose **Tabular** layout and **Portrait** orientation.
- Choose the **Bold** style.
- Type **Book List** as the report's title.

2. Create Queries:

Query 1: List all of the books that are published by **Prentice Hall** using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the **ListPrice** field. Save this query as **Prentice Hall Books**.

Query 2: List all of the books whose **ISBN** begins with a 0 using the **ISBN**, **Title**, **Author**, **Year**, and **ListPrice** fields of the **Books** table. Save this query as **ISBN Starting with 0**.

Query 3: List all of the books that are published **after** 1995 **or** whose price is greater than 25 dollars using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the published date. Save this query as **New Books**.

Query 4: List all of the products whose **UnitsOnOrder** is greater than 20 using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Sort the query in **ascending** order according to the **UnitPrice** field. Save this query as **Big Order**.

Query 5: List all of the products whose **UnitsOnOrder** is equal to 0 **and** whose **UnitPrice** is between **50** and **100** dollars using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Save this query as **Products Costing Between 50 and 100**.

Figure 6.87 - Seventh test page of text, skewed 2° counter-clockwise.

1. Create a Report:

Use the Report Wizard and choose the **Books** table to **create** a report having the following requirements:

- Select the **Author**, **Year**, **ListPrice** and **Publisher** fields.
- Do not choose any grouping levels.
- Sort the report by 1) the **Year** field in **descending** order and 2) the **ListPrice** field in **ascending** order.
- Choose **Tabular** layout and **Portrait** orientation.
- Choose the **Bold** style.
- Type **Book List** as the report's title.

2. Create Queries:

Query 1: List all of the books that are published by **Prentice Hall** using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the **ListPrice** field. Save this query as **Prentice Hall Books**.

Query 2: List all of the books whose **ISBN** begins with a 0 using the **ISBN**, **Title**, **Author**, **Year**, and **ListPrice** fields of the **Books** table. Save this query as **ISBN Starting with 0**.

Query 3: List all of the books that are published **after** 1995 **or** whose price is greater than 25 dollars using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the published date. Save this query as **New Books**.

Query 4: List all of the products whose **UnitsOnOrder** is greater than 20 using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Sort the query in **ascending** order according to the **UnitPrice** field. Save this query as **Big Order**.

Query 5: List all of the products whose **UnitsOnOrder** is equal to 0 **and** whose **UnitPrice** is between **50** and **100** dollars using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Save this query as **Products Costing Between 50 and 100**.

Figure 6.88 - Seventh test page of text, skewed 2° clockwise.

1. Create a Report:

Use the Report Wizard and choose the **Books** table to **create** a report having the following requirements:

- Select the **Author**, **Year**, **ListPrice** and **Publisher** fields.
- Do not choose any grouping levels.
- Sort the report by 1) the **Year** field in **descending** order and 2) the **ListPrice** field in **ascending** order.
- Choose **Tabular** layout and **Portrait** orientation.
- Choose the **Bold** style.
- Type **Book List** as the report's title.

2. Create Queries:

Query 1: List all of the books that are published by **Prentice Hall** using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the **ListPrice** field. Save this query as **Prentice Hall Books**.

Query 2: List all of the books whose **ISBN** begins with a 0 using the **ISBN**, **Title**, **Author**, **Year**, and **ListPrice** fields of the **Books** table. Save this query as **ISBN Starting with 0**.

Query 3: List all of the books that are published **after** 1995 **or** whose price is greater than 25 dollars using the **Title**, **Year**, **ListPrice**, and **Publisher** fields of the **Books** table. Sort the query in **descending** order according to the published date. Save this query as **New Books**.

Query 4: List all of the products whose **UnitsOnOrder** is greater than 20 using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Sort the query in **ascending** order according to the **UnitPrice** field. Save this query as **Big Order**.

Query 5: List all of the products whose **UnitsOnOrder** is equal to 0 **and** whose **UnitPrice** is between **50** and **100** dollars using the **ProductName**, **UnitPrice**, and **UnitsOnOrder** fields of the **Products** table. Save this query as **Products Costing Between 50 and 100**.

1

Figure 6.89 - Seventh test page of text, skewed 6° clockwise.

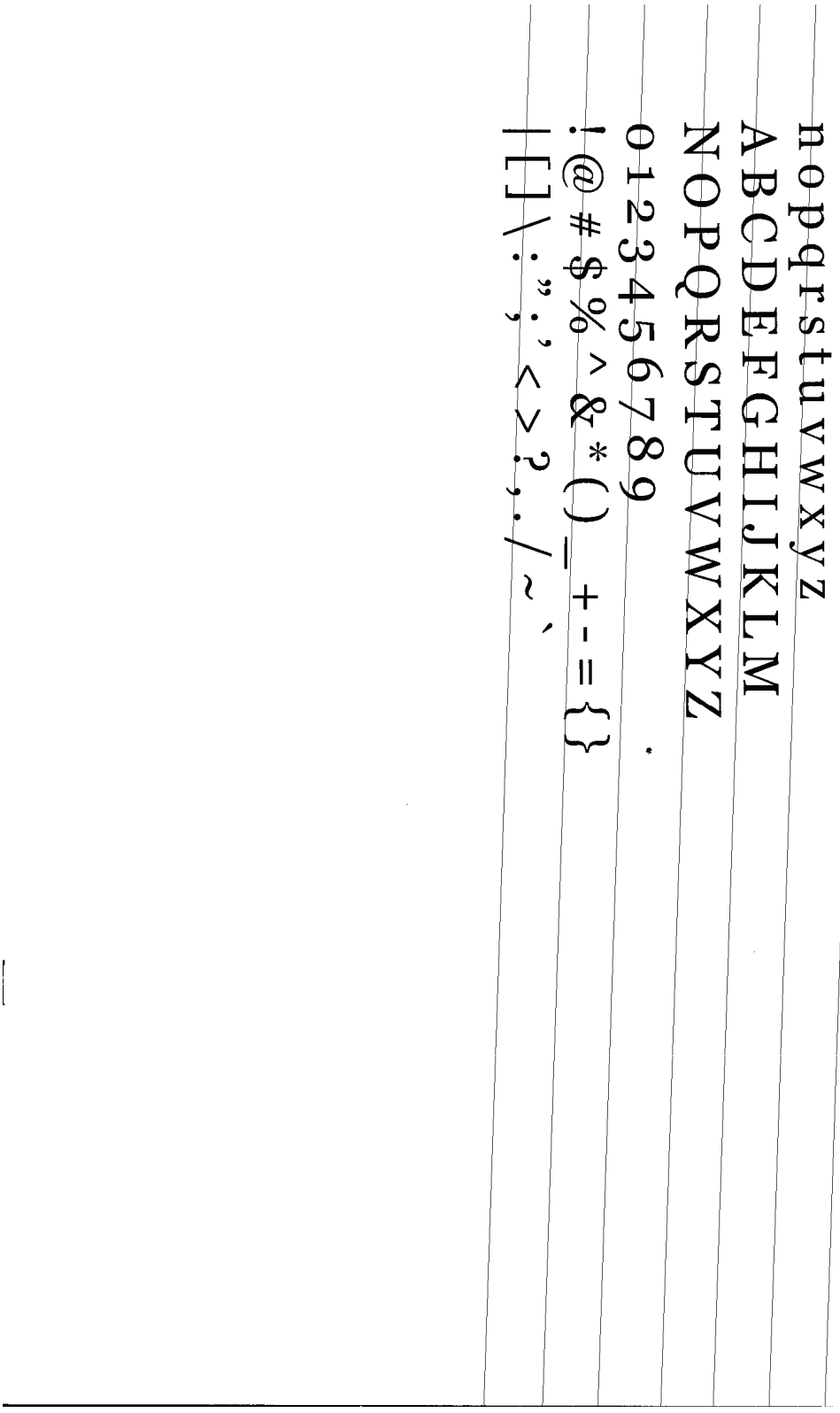


Figure 6.90 - Eighth test page of text, not skewed.

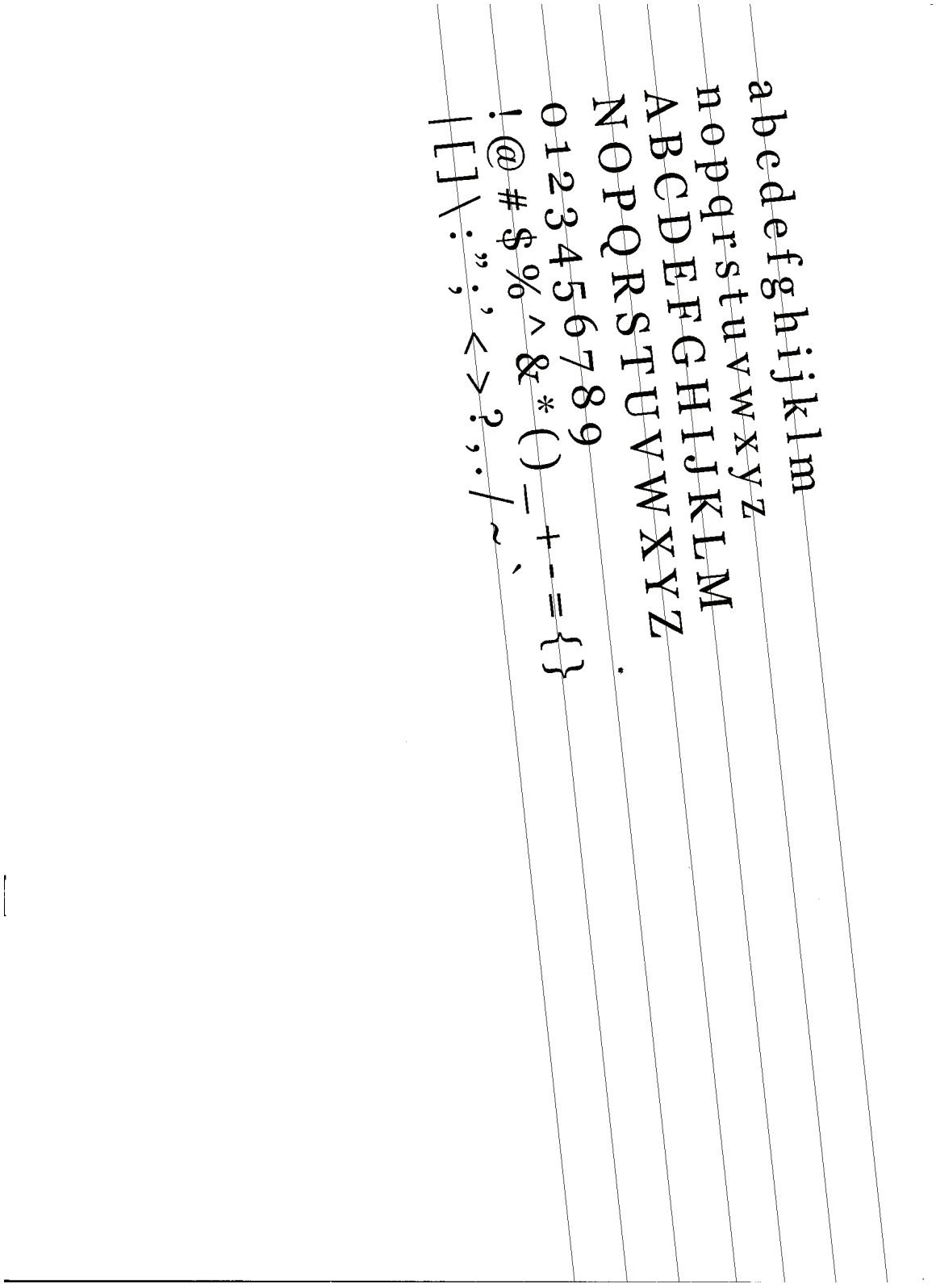


Figure 6.91 - Eighth test page of text, skewed 6° counter-clockwise.

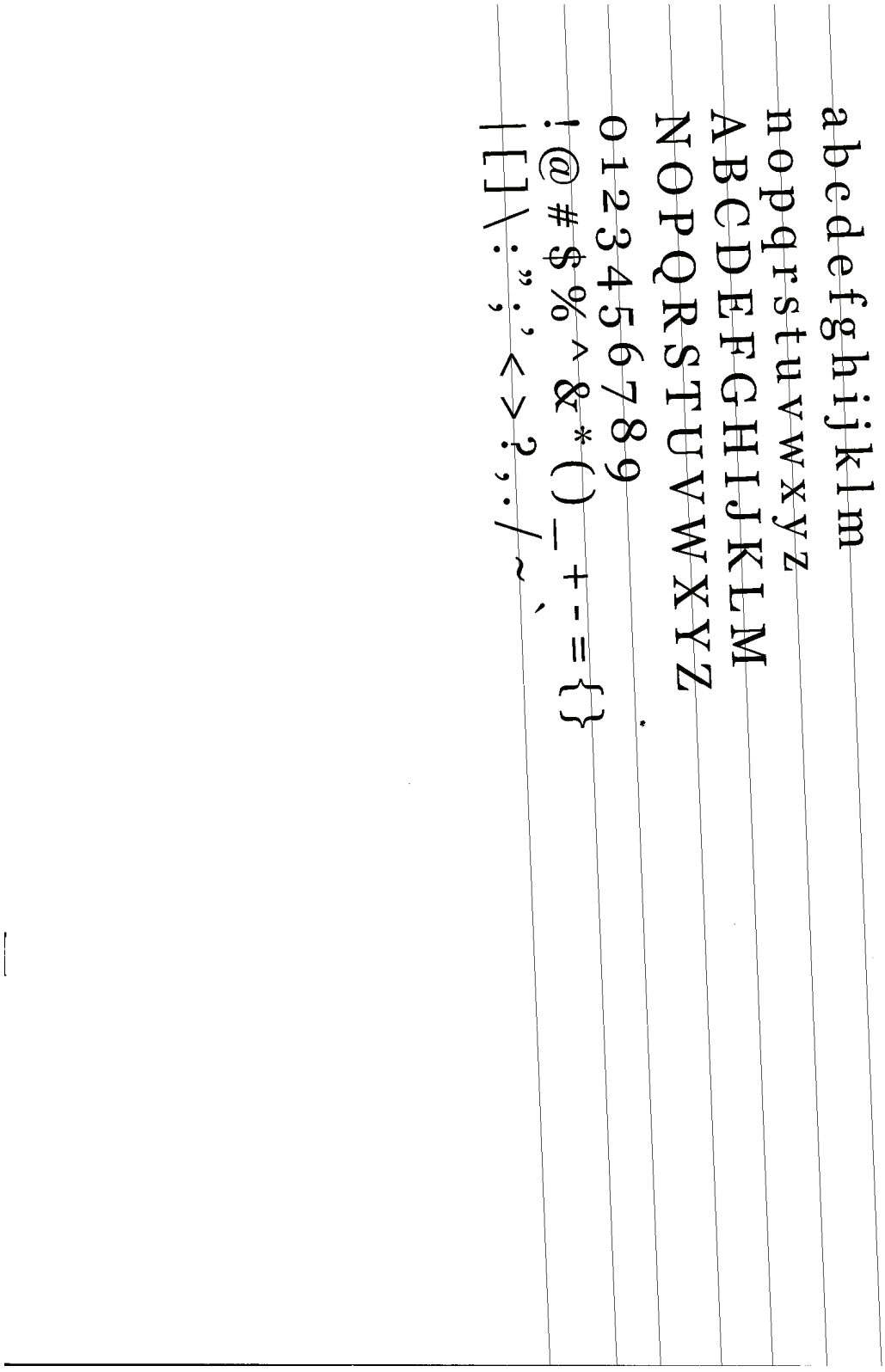


Figure 6.92 - Eighth test page of text, skewed 2° counter-clockwise.

Figure 6.93 - Eighth test page of text, skewed 2° clockwise.

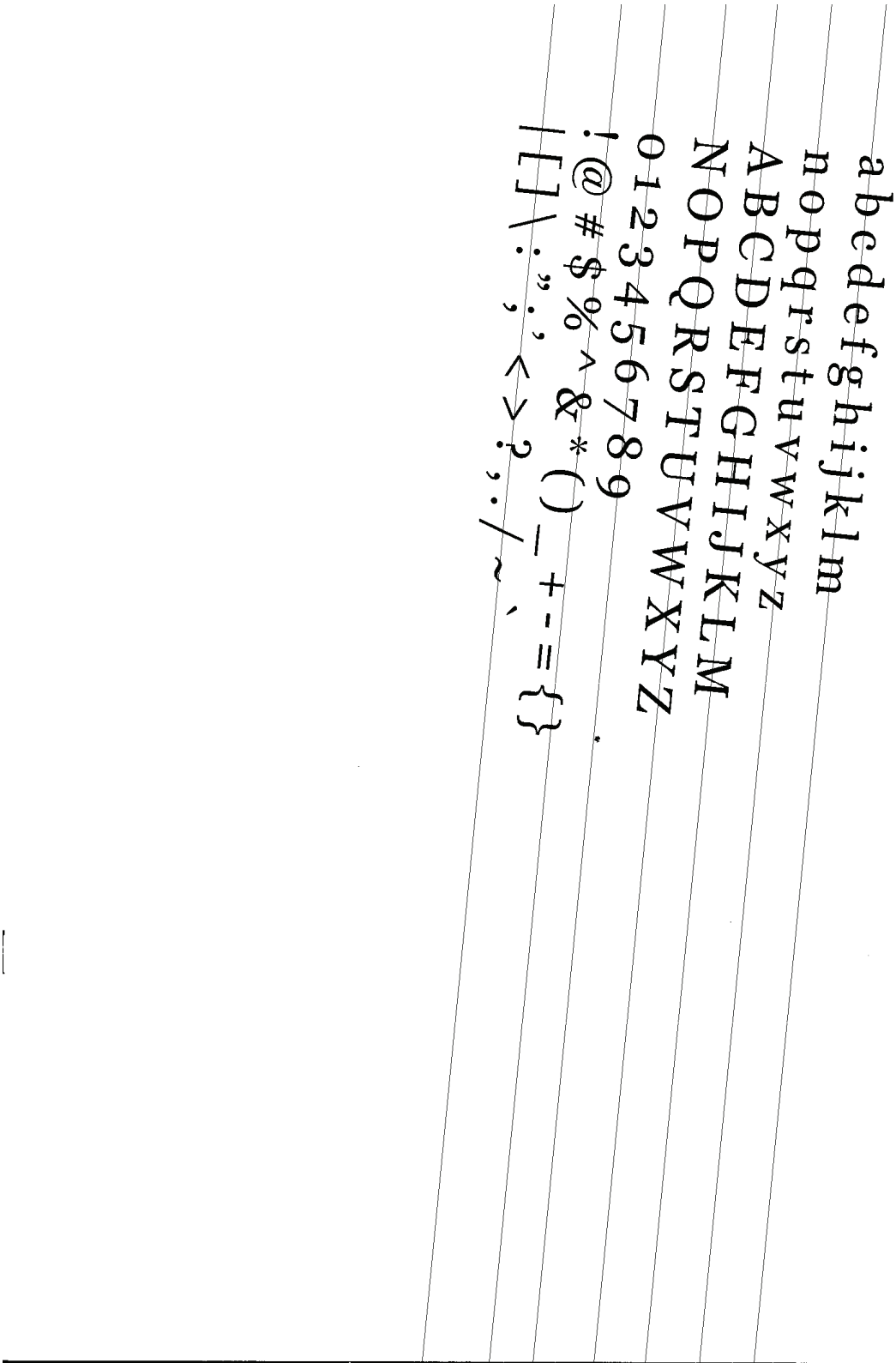


Figure 6.94 - Eighth test page of text, skewed 6° clockwise.

6.2.2 Moment Value Comparison of Font Name and Font Size Combinations

Recall from subsection 3.6.2 in chapter 3, the Character Database Creation (CDC) program adds to the Optical Character Recognition (OCR) database all of the printable characters from the U.S. English keyboard for each font name and font size combination image fed to the CDC program. Figure 6.95 presents an image containing all of the printable characters from the U.S. English keyboard for the font name and font size combination of Arial 36 point.

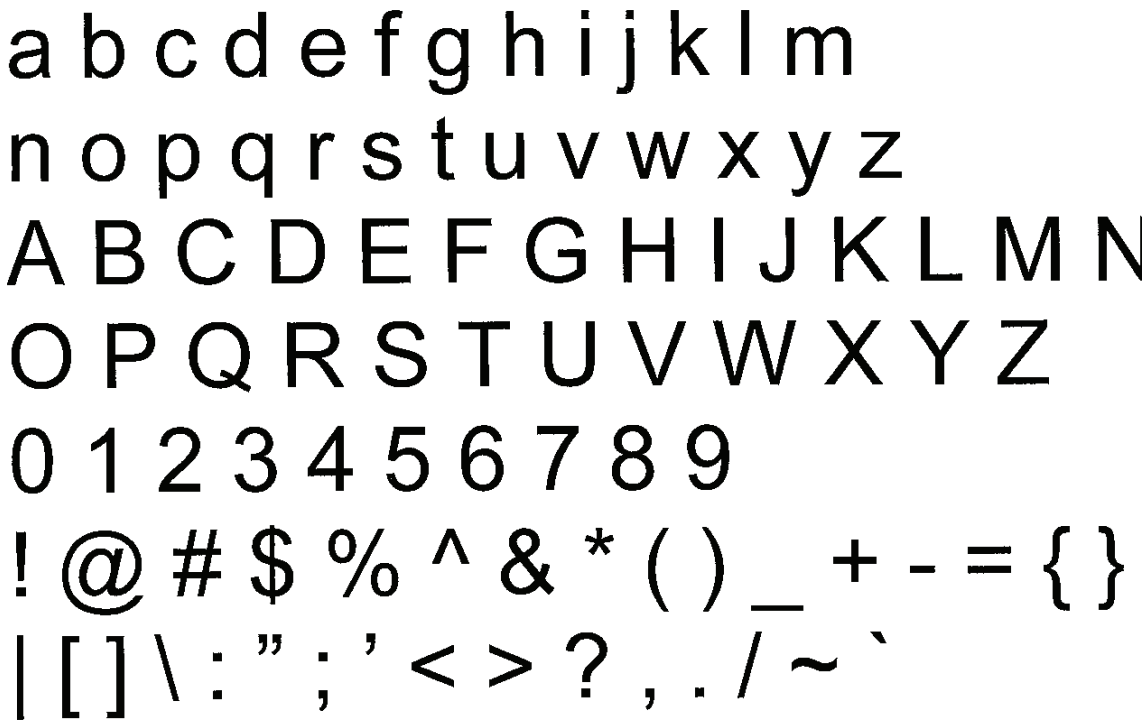


Figure 6.95 – Set of all printable characters with font name and size of Arial 36 point.

The CDC program computes the 19 measure values for each character in the font name and font size combination's image. The measure values are the character's axis

aligned area, the character's minimum area rectangle area, the 7 Hu moments, and the 10 Flusser-Suk moments, defined in subsection 3.6.1 of chapter 3. The CDC program creates a record in the OCR database's Characters table for each character in the font name and font size combination's image. The information stored in the character's record is the character itself, its font name, its font size and its 19 measure values. After processing all of the characters in the font name and font size combination's image, the CDC program computes for the font name and font size combination: the minimum value, maximum value and average value for each of the 19 measures. The CDC program creates a record in the OCR database's Properties table for the font name and font size combination. The information stored in the font name and font size combination's record is the font name, font size, and the minimum value, maximum value and average value for each of the 19 measures. The 28 font name and font size combinations stored in the OCR database are [Arial, Courier New, Georgia, Times New Roman] x [12 pt., 18 pt., 24 pt., 30 pt., 36 pt., 42 pt., 48 pt.]

The minimum, maximum and average values for each measure and font name and font size combination allow a visual comparison of the range of values for the 28 font name and font size combinations for the measure. These values are in the OCR database's Properties table. Retrieve and place all of the data from the OCR database's Properties table into an Excel spreadsheet. A measure's minimum value, maximum value and average value for all 28 font name and font size combinations form a stock type chart. A stock type chart requires a series of three values for a group of items. The three stock values are the high, low and close values. In this study, the items are the font name and font size combinations and the series of three values are the measure's maximum

value, minimum value and average value. This type of chart allows an easy comparison of the measure's range of values for the entire set of font name and font size combinations. The chart in figure 6.96 is a comparison of the axis aligned area range of values for all 28 font name and font size combinations. The chart in figure 6.97 is a comparison of the minimum area rectangular area range of values for all 28 font name and font size combinations. Each chart in figures 6.98 through 6.104 is a comparison of one of the Hu moment's range of values for all 28 font name and font size combinations. Each chart in figures 6.105 through 6.114 is a comparison of one of the Flusser-Suk moment's range of values for all 28 font name and font size combinations.

A short range for a particular moment of a font name and size combination signifies that the moment values for the font name and font size combination's characters are close together. Therefore, the moment may not be useful in distinguishing between one character and another within the font name and font size combination. A long range for a particular moment of a font name and font size combination signifies that the moment values for the font name and font size combination's characters are further apart. Therefore, the moment may be useful in distinguishing between one character and another within the font name and size combination.

A short overlap of two font name and font size combinations for a particular moment indicates that the number of characters sharing similar moment values between the two font name and font size combinations is small. Therefore, the moment may be useful in distinguishing between characters of the two font name and font size combinations. A long overlap of two font name and font size combinations for a particular moment indicates that the number of characters sharing similar moment values

between the two font name and size combinations is large. Therefore, the moment may not be useful in distinguishing between characters of the two font name and font size combinations.

An examination of the charts for the axis aligned area and minimum area rectangle area measures (figures 6.96 and 6.97) reveals that the ranges within a particular font name increase as the font size increases. The Courier New font name has ranges that are smaller in size in relation to the other three font names. The ranges for each font name follow the same pattern and size possibly making them little use for character recognition. Examining the charts for the Hu moments (figures 6.98 through 6.104) shows that the font name and font size combination range shortens for each succeeding Hu moment. The Arial font name ranges still have significant length for Hu moments 1 through 6. The one font name and size combination that stands out for the Hu moments is Times New Roman 12 point. The range for Times New Roman 12 point remains long for all 7 Hu moments while the ranges for all of the other font name and size combinations shorten from Hu moment 1 to 7. A final interesting observation is the Courier New font name ranges suddenly change from short to long between Hu moment 6 and 7. Examining the charts for the Flusser-Suk moments (figures 6.105 through 6.114) reveals that the range for the Arial and Georgia font names shortens for each succeeding Flusser-Suk moment. The range for the Courier New and Times New Roman font names remain long for all ten of the Flusser-Suk moments. The next chapter presents this dissertation's conclusions and future work.

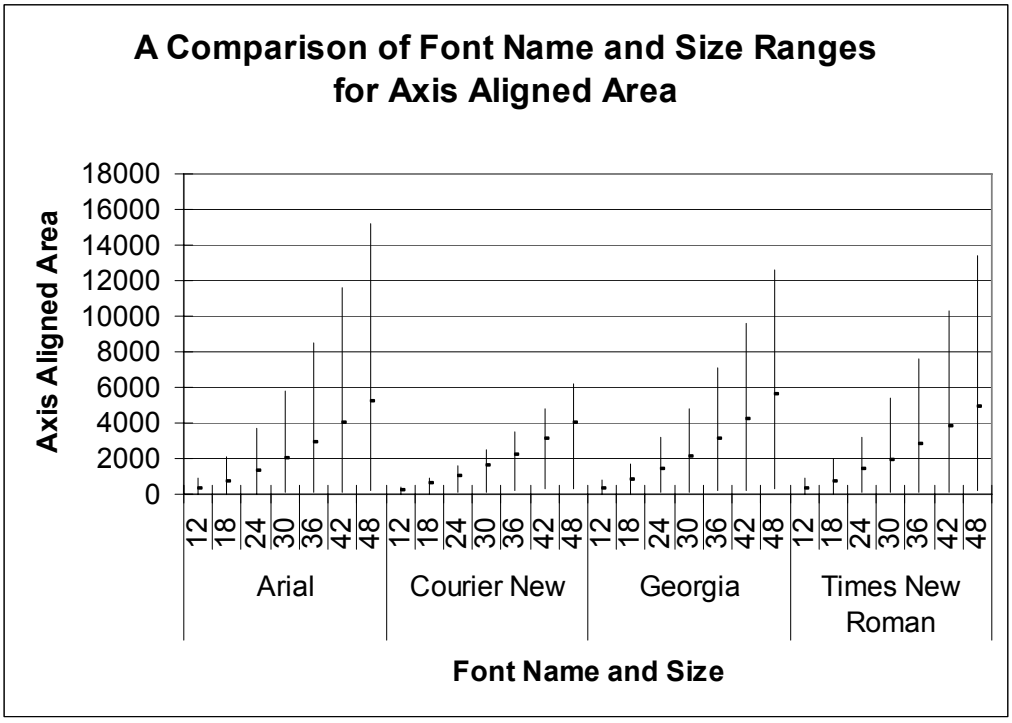


Figure 6.96 - Axis aligned area measure ranges for each font name and size combination.

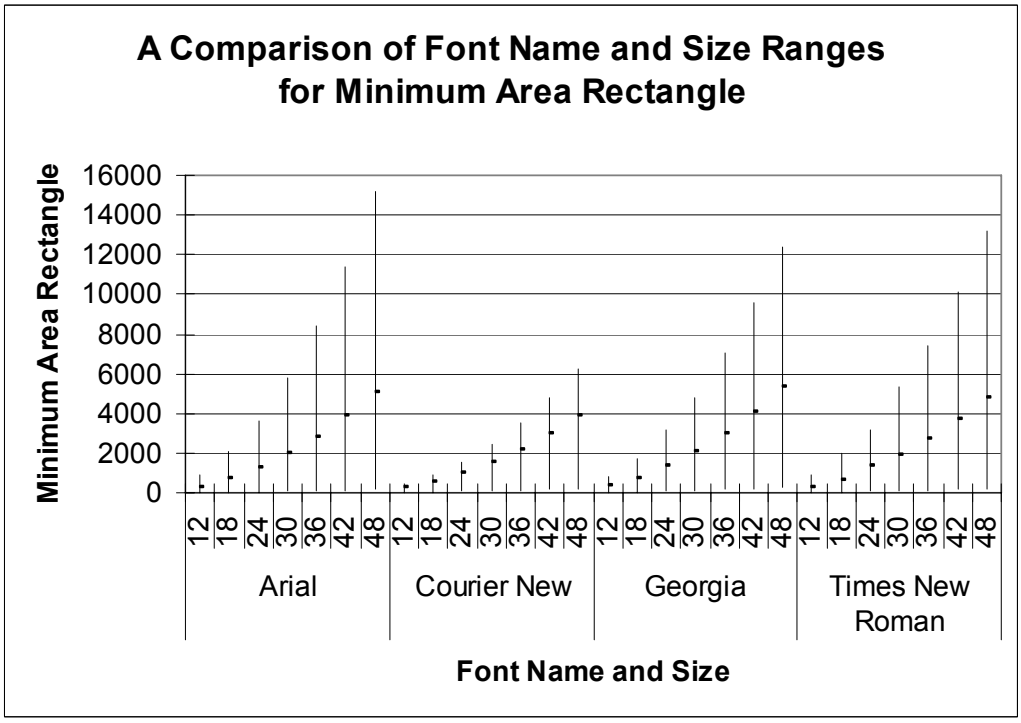


Figure 6.97 - Minimum area rectangle measure ranges
for each font name and size combination.

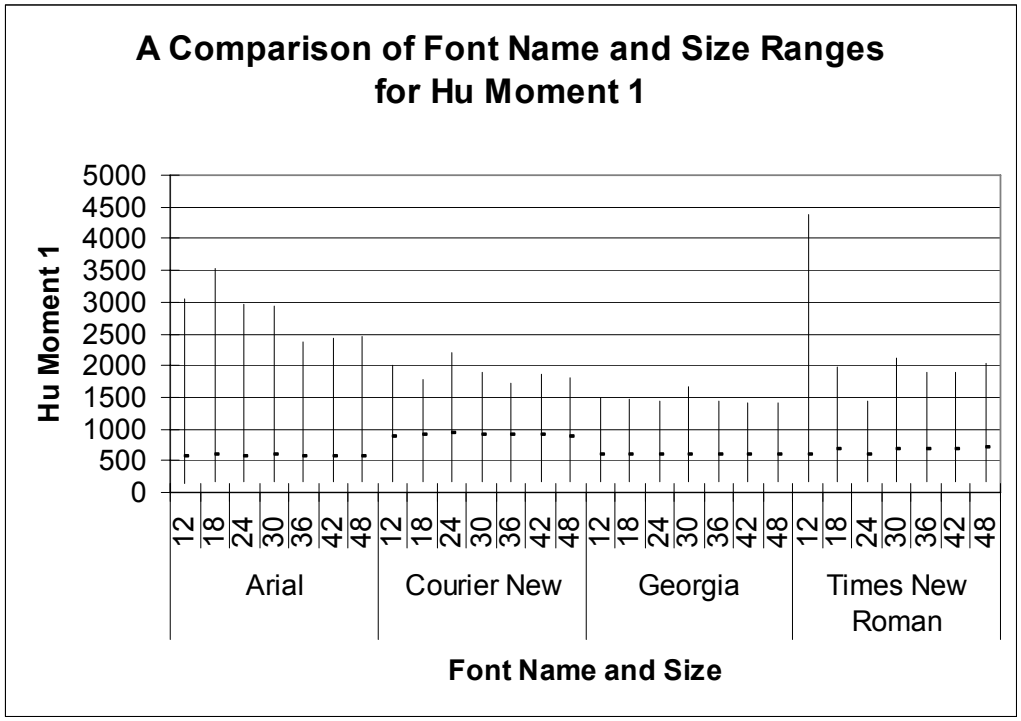


Figure 6.98 - Hu moment 1 ranges for each font name and size combination.

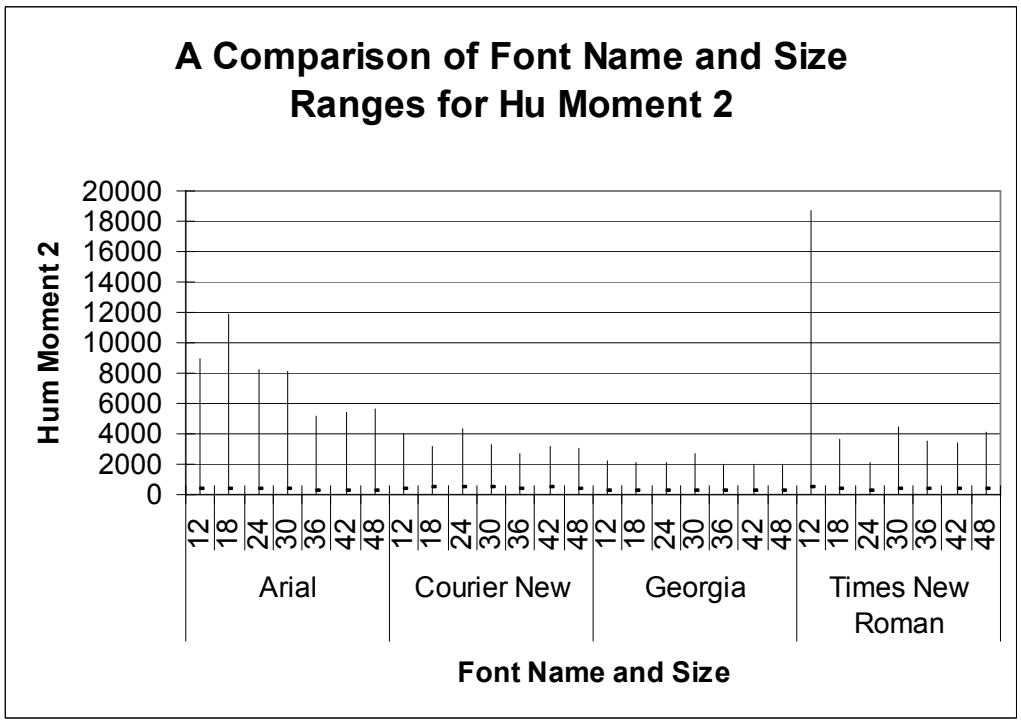


Figure 6.99 - Hu moment 2 ranges for each font name and size combination.

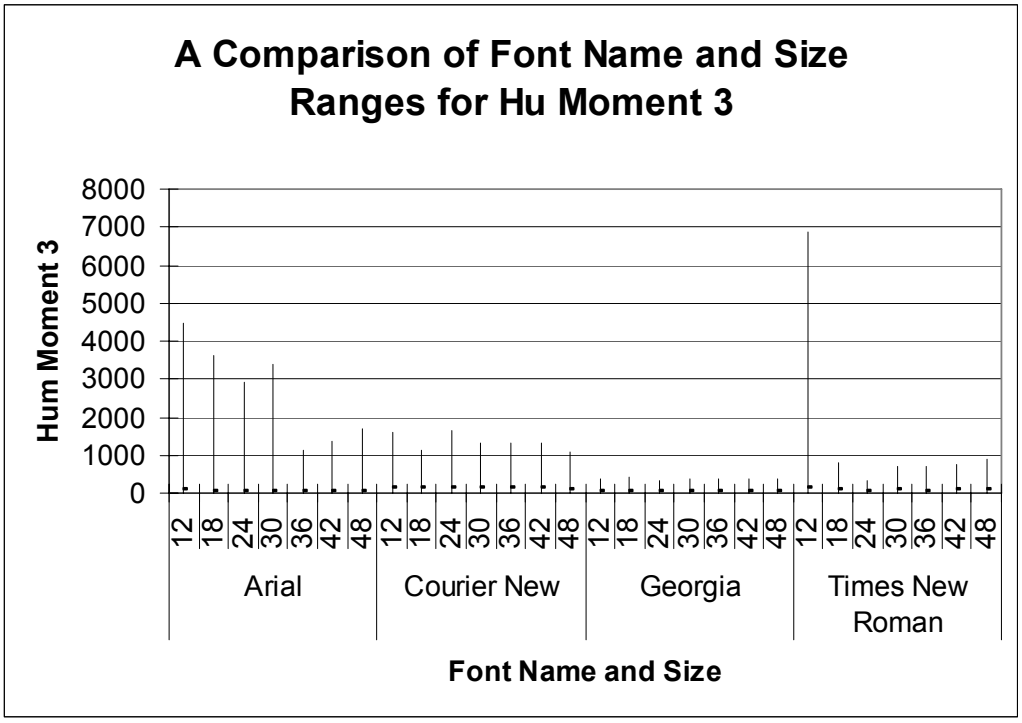


Figure 6.100 - Hu moment 3 ranges for each font name and size combination.

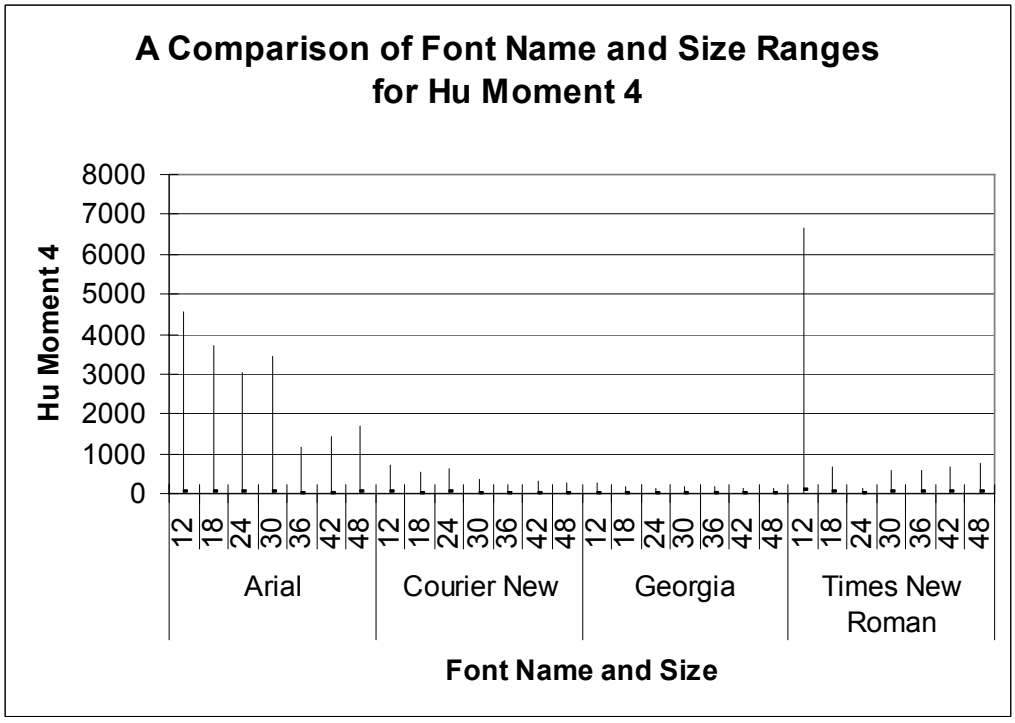


Figure 6.101 - Hu moment 4 ranges for each font name and size combination.

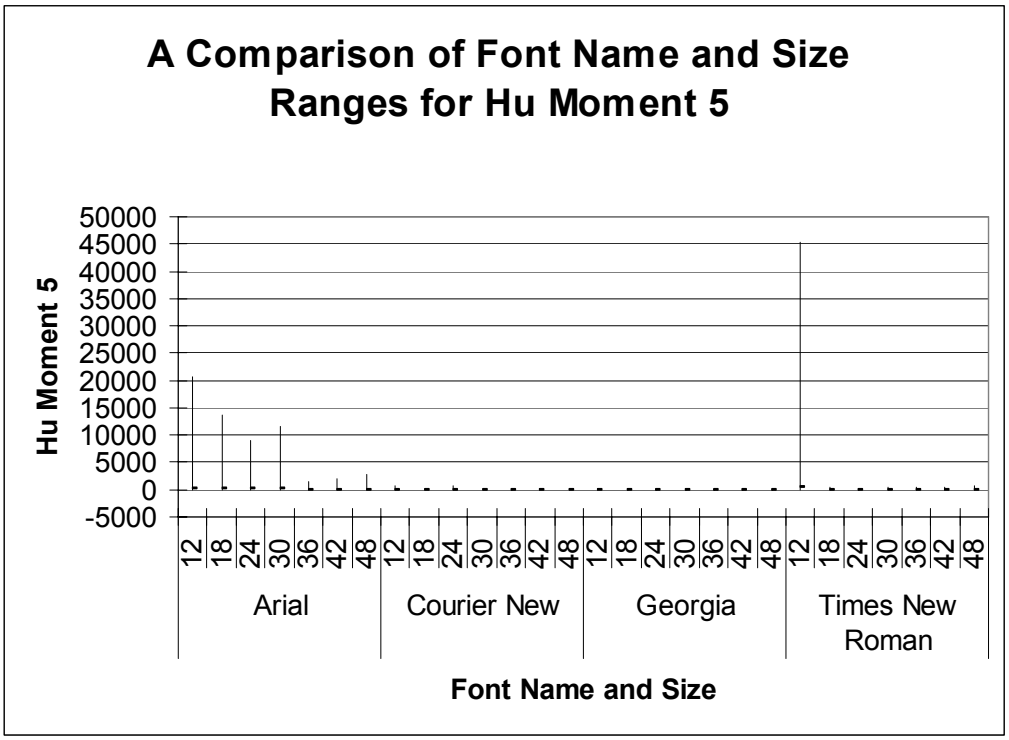


Figure 6.102 - Hu moment 5 ranges for each font name and size combination.

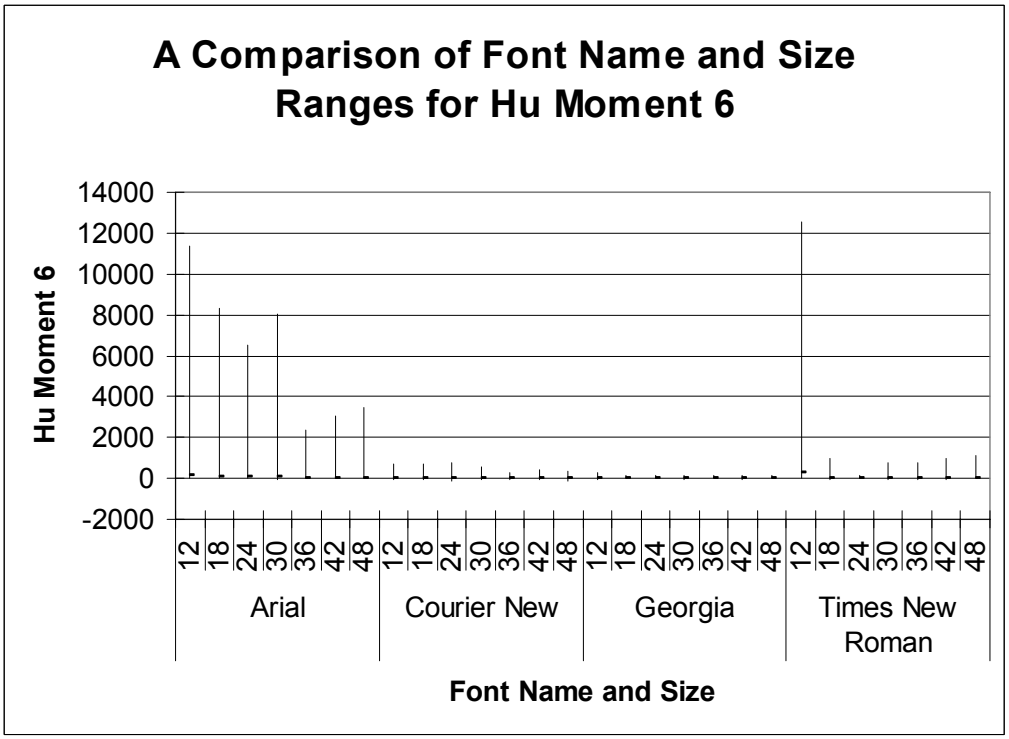


Figure 6.103 - Hu moment 6 ranges for each font name and size combination.

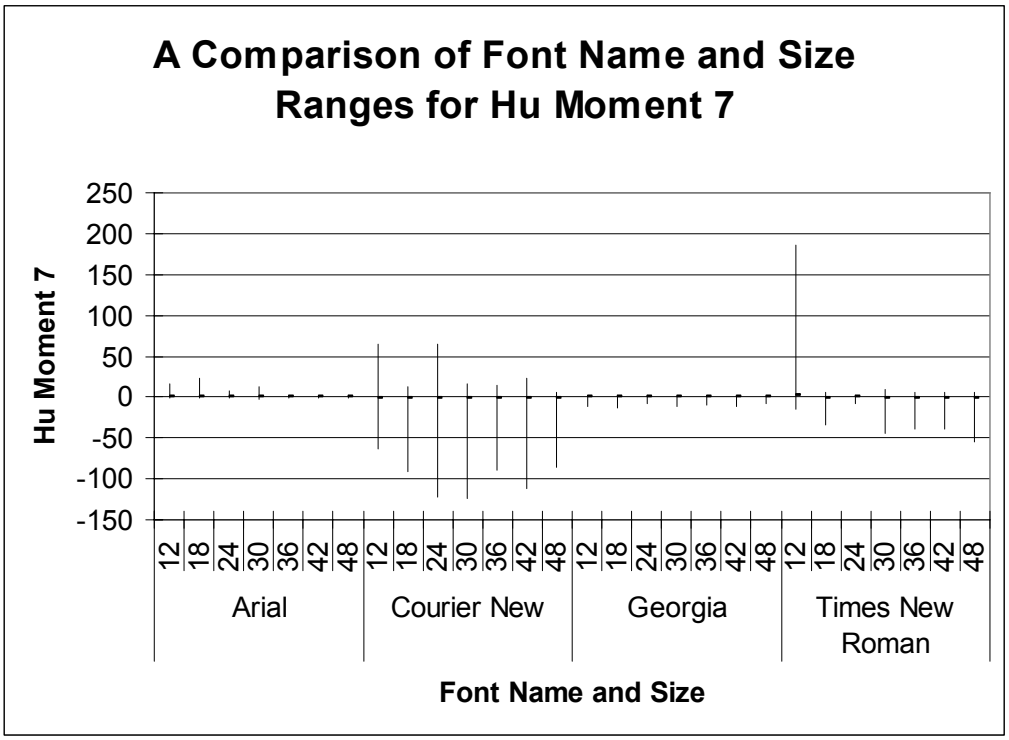


Figure 6.104 - Hu moment 7 ranges for each font name and size combination.

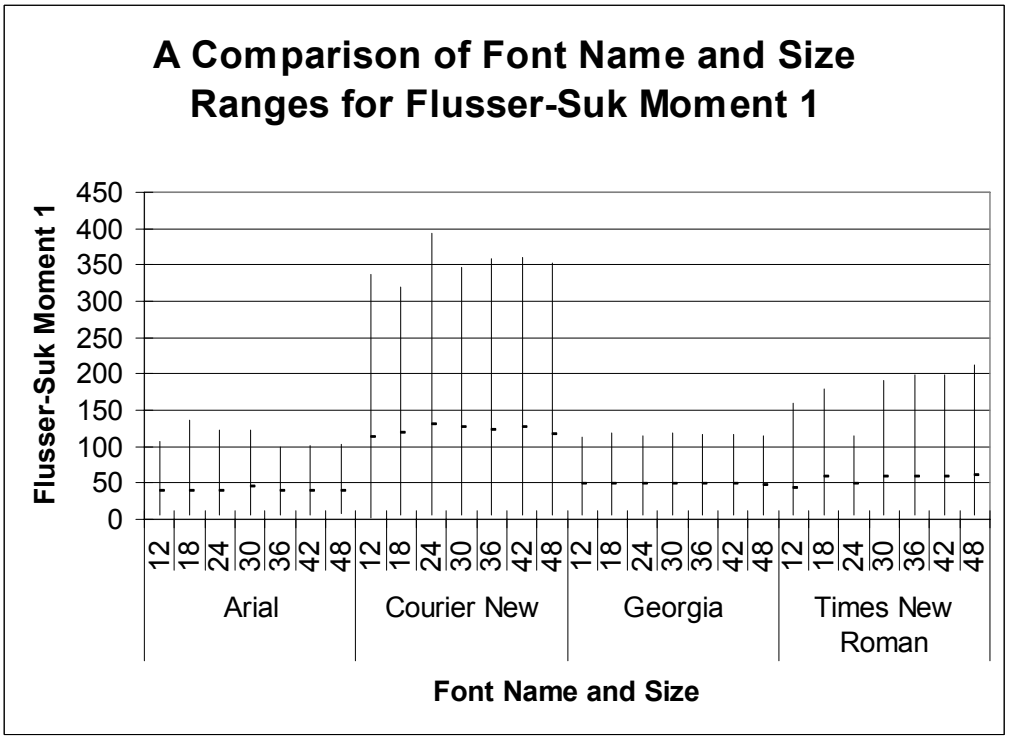


Figure 6.105 - Flusser-Suk moment 1 ranges for each font name and size combination.

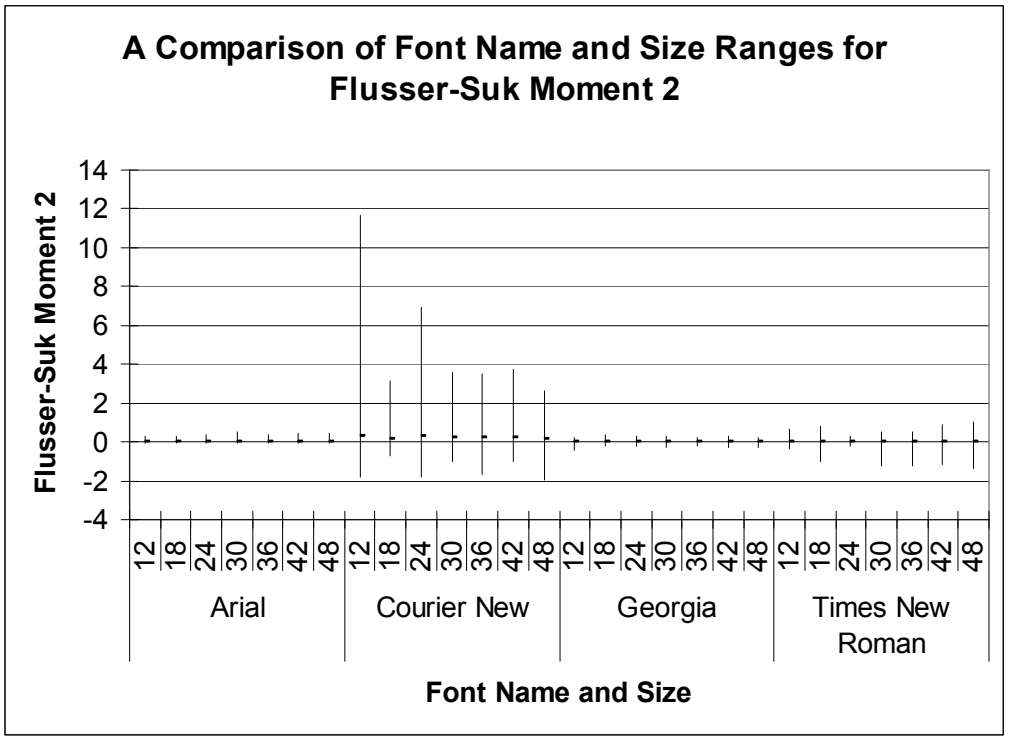


Figure 6.106 - Flusser-Suk moment 2 ranges for each font name and size combination.

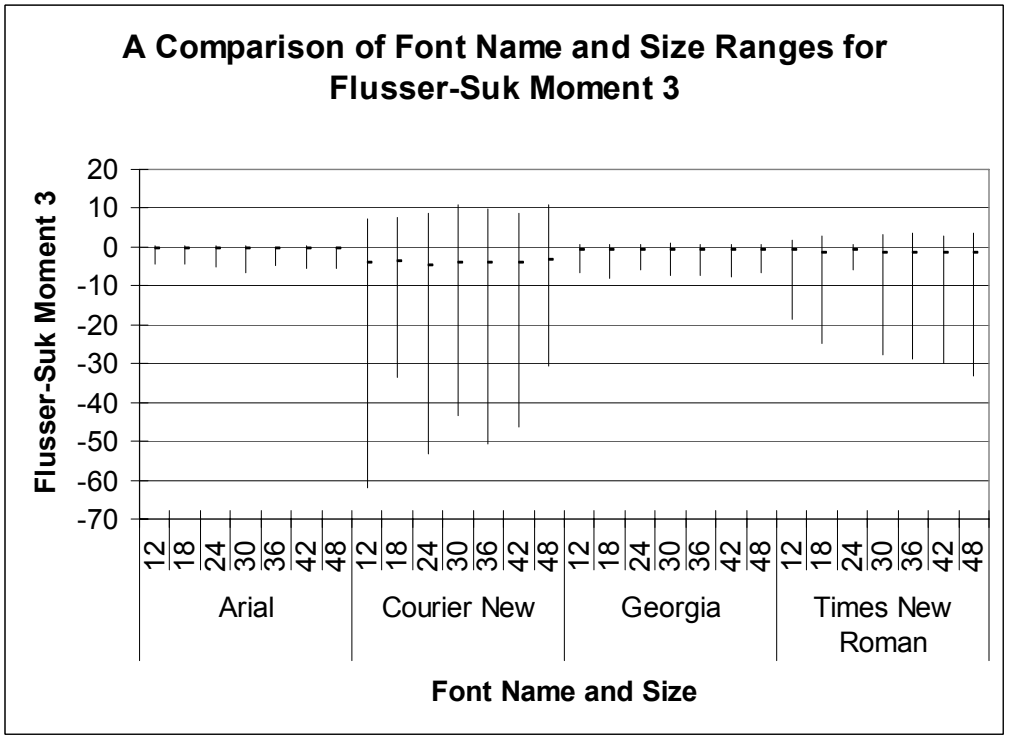


Figure 6.107 - Flusser-Suk moment 3 ranges for each font name and size combination.

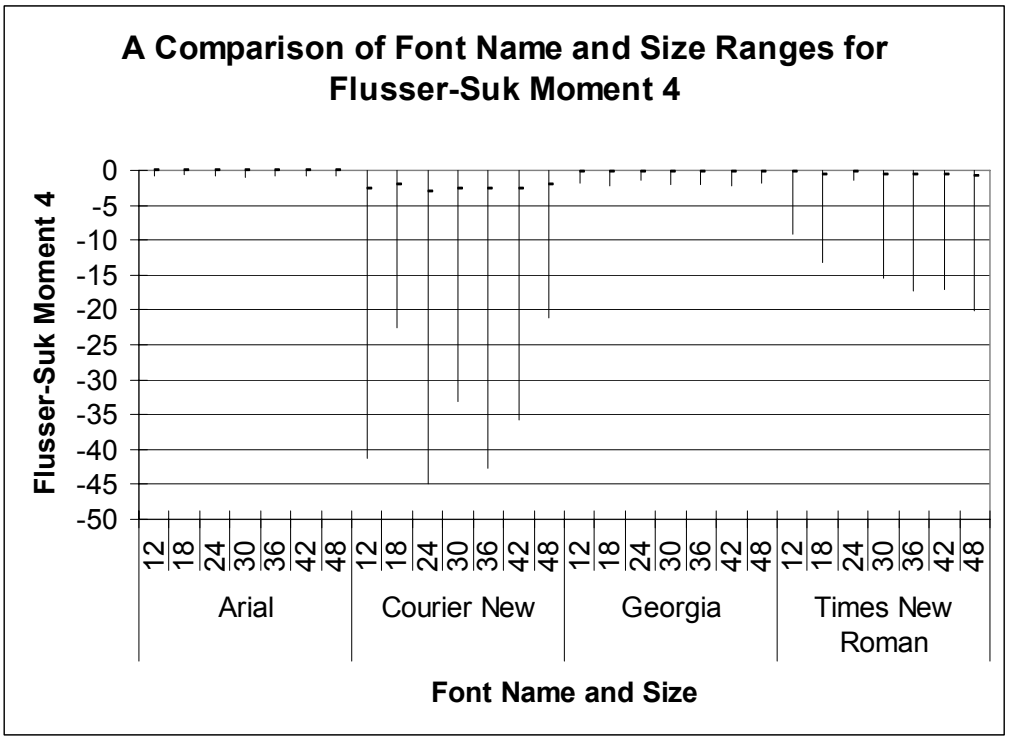


Figure 6.108 - Flusser-Suk moment 4 ranges for each font name and size combination.

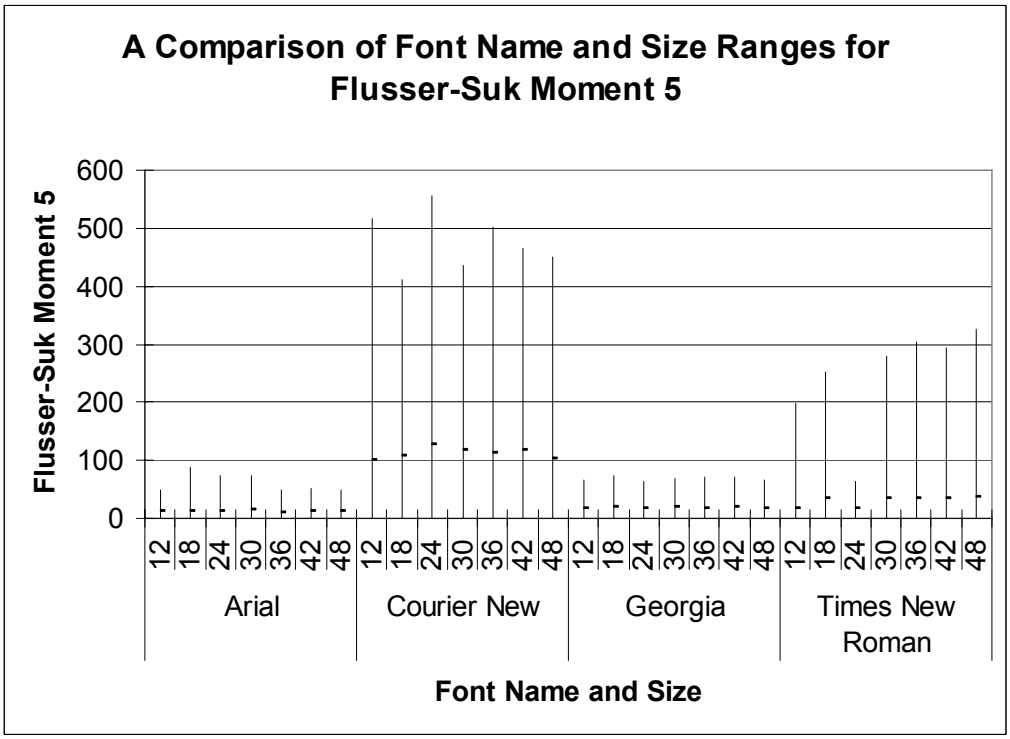


Figure 6.109 - Flusser-Suk moment 5 ranges for each font name and size combination.

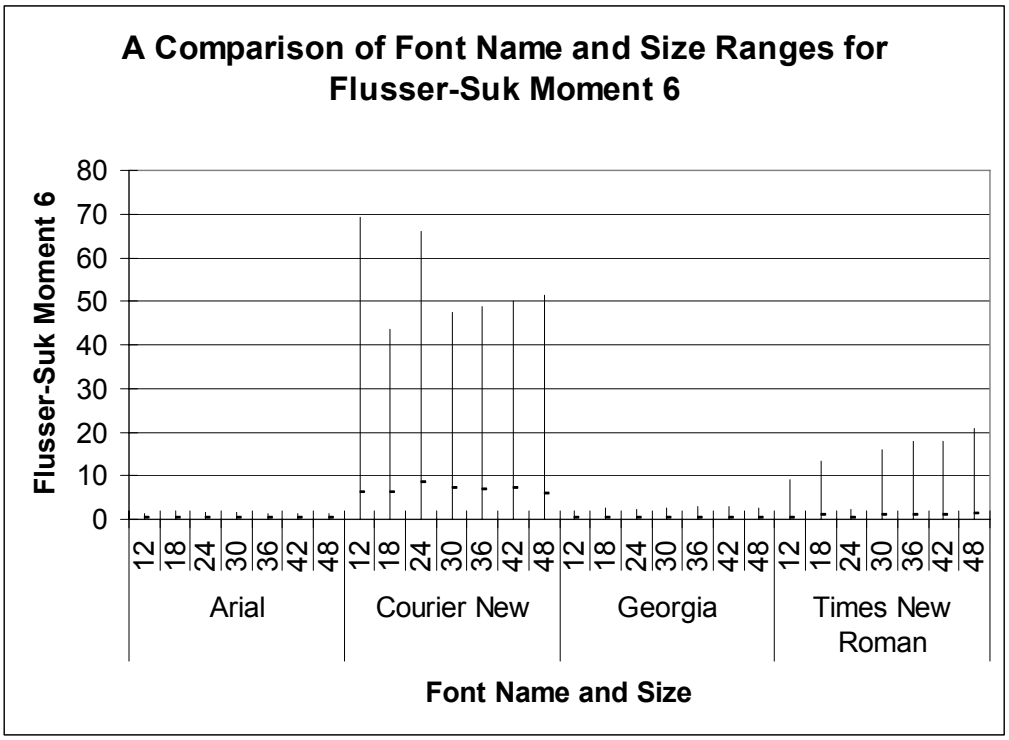


Figure 6.110 - Flusser-Suk moment 6 ranges for each font name and size combination.

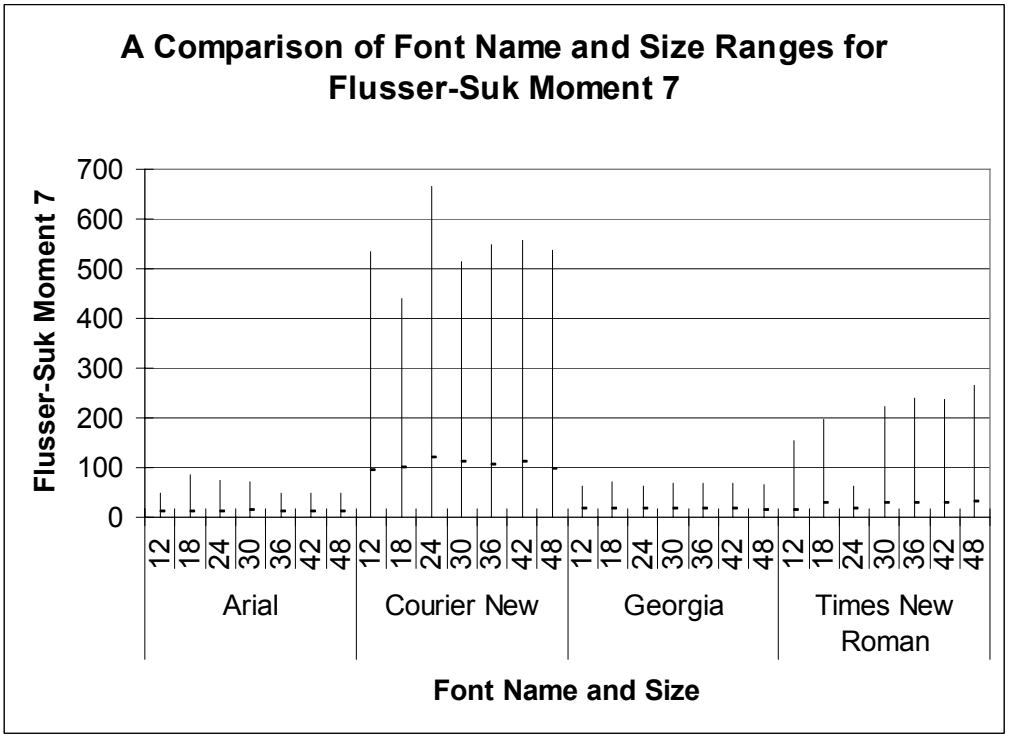


Figure 6.111 - Flusser-Suk moment 7 ranges for each font name and size combination.

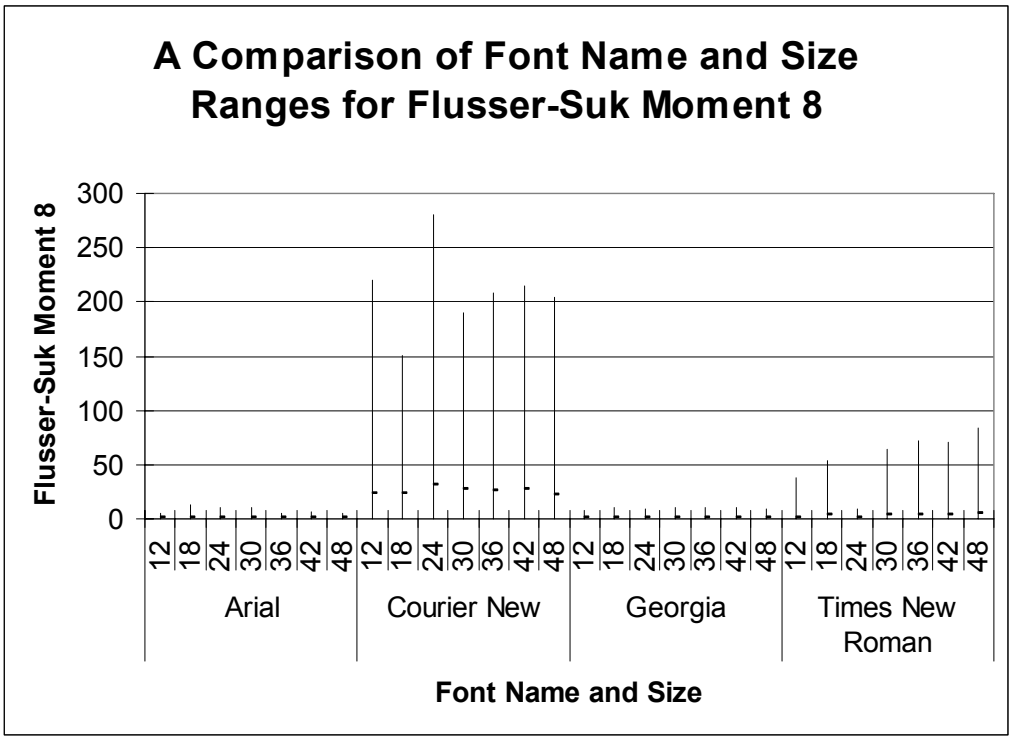


Figure 6.112 – Flusser-Suk moment 8 ranges for each font name and size combination.

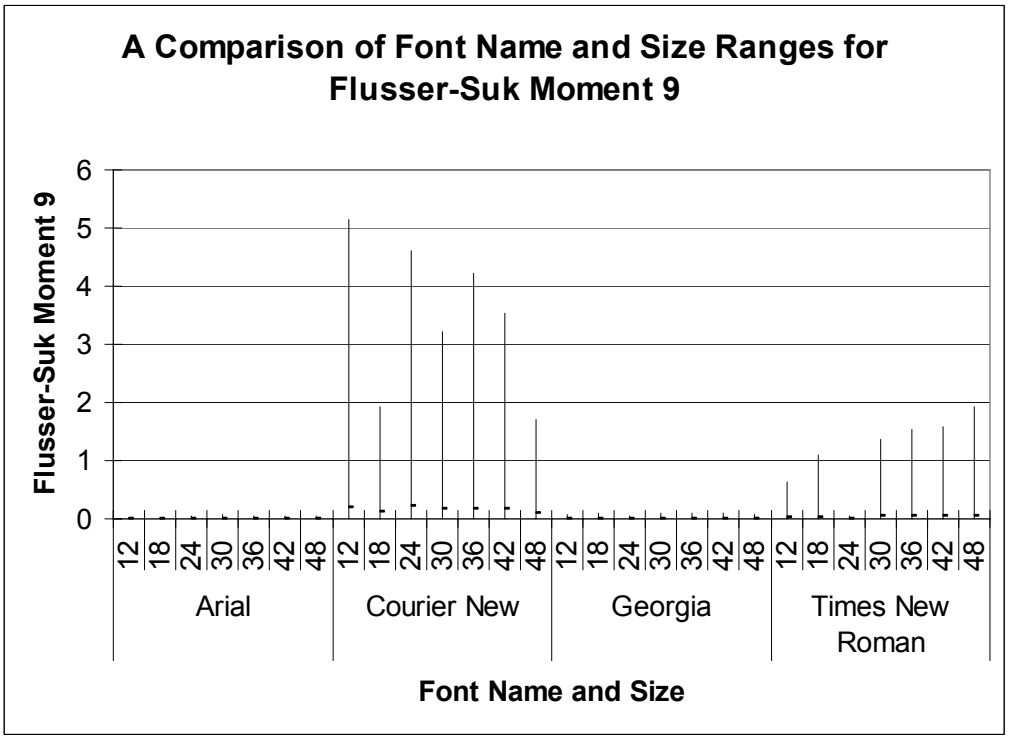


Figure 6.113 - Flusser-Suk moment 9 ranges for each font name and size combination.

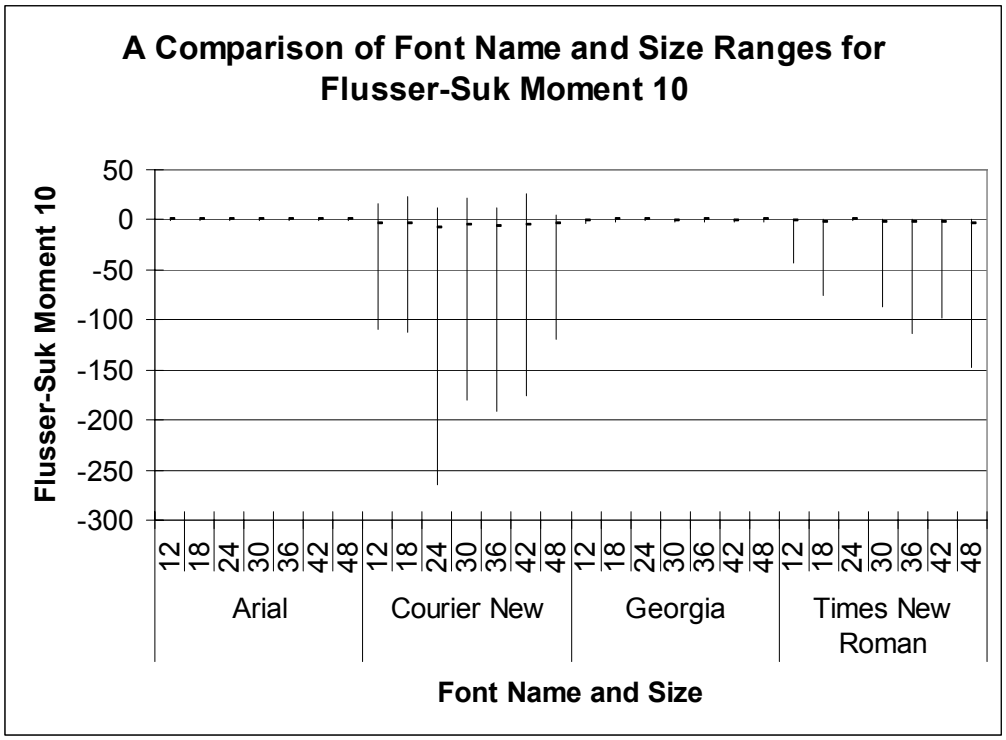


Figure 6.114 - Flusser-Suk moment 10 ranges for each font name and size combination.

Chapter 7 Conclusions and Future Work

The signature recognition problem is the first of the two problems addressed in this dissertation. Scan a signature into an image file, a digitized grid. Apply a standard thinning algorithm to the signature to create a version of the signature in which the lines and curves of the signature are one pixel wide. Treat the one pixel wide version of the signature as a path. Evolving agents (finite state machines) dedicated to particular paths is a means to recognizing signatures with the agent itself encapsulating the structure of some or all of the curves of a signature. The signature recognition genetic algorithm can evolve a single or multiple agents for a signature. The five experiments of this problem studied the feasibility of agents as descriptors for signatures.

The agent is a finite state machine with 32 states, 3 inputs and 3 outputs. The inputs are information about the environment in front of the agent. The outputs are the movements the agent performs. The signature recognition genetic algorithm breeds agents to recognize a particular path. The fitness of an agent is the amount of the path consumed by the agent. The genetic algorithm processes one generation after another until the current generation produces at least one agent that consumes the entire path. In this case, successful, the genetic algorithm writes to the output file all agents that consume the entire path. Otherwise, unsuccessful, the genetic algorithm terminates after the completion of generation 125. The signature recognition genetic algorithm is a part of the signature recognition program. The signature recognition program acts as the driver program for the genetic algorithm. Each execution of the genetic algorithm for a path is a run. The signature recognition program executes 100 separate runs of the genetic algorithm in order to produce a set of consistent results for the path.

The five experiments of the signature recognition problem utilize one of two sets of data. The first data set is the set of 10,000 non-crossover paths representing a large number of types of handwritten signature shapes. A non-crossover path is a path in which no part of the path can crossover any other part of the path. The non-crossover path lengths range from 13 to 2644. A count of the number of paths of each length shows that the counts form a bell curve centered at the path length range of 601 to 800. A path's winding number is the number of right angle turns in the path which is a measure of the path's geometry. The non-crossover path winding numbers range from 2 to 84. A count of the number of paths of each winding number shows that the counts form a bell curve centered at the winding number range of 11 to 20. The results of the path counts for the path lengths and winding numbers of the non-crossover paths reveals that the path creation process supplied paths with a good sample of different lengths and geometries.

The second data set is the set of 10,000 crossover paths representing a large number of types of handwritten signature shapes. A crossover path is a path in which any part of the path can crossover any other part of the path. The crossover path lengths range from 13 to 2644. A count of the number of paths of each length shows that the path counts range from 0 to 11 with an average path count of 3.86. Forty-four path lengths have a path count of 0. The path counts over the path length range are well distributed providing a good distribution of lengths for the crossover paths. A path's winding number is the number of right angle turns in the path which is a measure of the path's geometry. The crossover path winding numbers range from 0 to 65. A count of the number of paths of each winding number reveals that the path counts from winding number 0 to 44 are within a narrow range of 145 to 237 paths. From winding number 45

to 65, the path counts decrease from 154 to 1. The path counts over the winding number range are well distributed providing a good distribution of geometries for the crossover paths.

The first signature recognition program experiment utilizes the non-crossover paths for its data set. The signature recognition genetic algorithm in the first experiment uses a fitness function that computes the fitness value for an agent with no concern to the order of consumption of the path squares. The average number of successful runs per path for the first experiment is 99.64 with a standard deviation of 2.45. All 10,000 non-crossover paths for the first signature recognition program experiment have at least one successful run. The average number of unsuccessful runs per path for the first experiment is 0.36 with 0 to 60 as the range of unsuccessful runs per path for the first experiment. The average final generation number per successful run for the first experiment is 3.73 with 0 to 125 as the range of final generation number per successful run for the first experiment. Compute the total number of agents found for each path across all of the successful runs of the path. Determine the number of agents of a path that are contiguous agents. A contiguous agent is an agent that consumes the path in full contiguous order, consuming each path square in order from beginning to end. The path's percentage of contiguous agents is the number of contiguous agents for a path divided by the path's total number of agents. A count of the number of paths of each contiguous agent percentage range shows that the counts form a bell curve centered at the percentage range of 30% to 40%. Sixty-three paths have a contiguous agent percentage of 0%.

The second signature recognition program experiment utilizes the non-crossover paths for its data set. The signature recognition genetic algorithm in the second experiment also uses the fitness function that computes the fitness value for the agent with no concern to the order of consumption of the path squares. After computing an agent's fitness value, the signature recognition program in the second experiment reorganizes an agent's states. The start state of all agents is always state 0. Determine the states in the agent that are reachable from the start state 0. The reorganization of the agent swaps pairs of states until the agent's n reachable states are states 0 through n-1 and the agent's non-reachable states are the remaining states in the agent. The average number of successful runs per path for the second experiment is 99.80 with a standard deviation of 1.66. All 10,000 non-crossover paths for the second signature recognition program experiment have at least one successful run. The average number of unsuccessful runs per path for the second experiment is 0.20 with 0 to 61 as the range of unsuccessful runs per path for the second experiment. The average final generation number per successful run for the second experiment is 3.63 with 0 to 125 as the range of final generation number per successful run for the second experiment. Compute each path's percentage of contiguous agents. A count of the number of paths of each contiguous agent percentage range shows that the counts form a bell curve centered at the percentage range of 30% to 40%. Sixty paths have a contiguous agent percentage of 0%.

The third signature recognition program experiment utilizes the non-crossover paths for its data set. The signature recognition genetic algorithm in the third experiment also uses the fitness function that computes the fitness value for the agent with no concern to the order of consumption of the path squares. After computing an agent's

fitness value, the signature recognition program in the third experiment reorganizes an agent's states. The start state of all agents is always state 0. Determine the states in the agent that are reachable from the start state 0. The reorganization of the agent swaps pairs of states until every other state is a reachable state and every other state is a non-reachable state. The average number of successful runs per path for the third experiment is 99.75 with a standard deviation of 1.99. All 10,000 non-crossover paths for the third signature recognition program experiment have at least one successful run. The average number of unsuccessful runs per path for the third experiment is 0.25 with 0 to 51 as the range of unsuccessful runs per path for the third experiment. The average final generation number per successful run for the third experiment is 3.67 with 0 to 125 as the range of final generation number per successful run for the third experiment. Compute each path's percentage of contiguous agents. A count of the number of paths of each contiguous agent percentage range shows that the counts form a bell curve centered at the percentage range of 30% to 40%. Seventy paths have a contiguous agent percentage of 0%.

The fourth signature recognition program experiment utilizes the crossover paths for its data set. The signature recognition genetic algorithm in the fourth experiment also uses the fitness function that computes the fitness value for the agent with no concern to the order of consumption of the path squares. The average number of successful runs per path for the fourth experiment is 96.91 with a standard deviation of 9.19. All 10,000 crossover paths for the fourth signature recognition program experiment have at least one successful run. The average number of unsuccessful runs per path for the fourth experiment is 3.09 with 0 to 86 as the range of unsuccessful runs per path for the fourth

experiment. The average final generation number per successful run for the fourth experiment is 7.71 with 0 to 125 as the range of final generation number per successful run for the fourth experiment. Compute each path's percentage of contiguous agents. A count of the number of paths of each contiguous agent percentage range shows that the counts form a bell curve centered at the percentage range of 50% to 60%. 3,378 paths have a contiguous agent percentage of 0%.

The fifth signature recognition program experiment utilizes the crossover paths for its data set. The signature recognition genetic algorithm in the fifth experiment uses a fitness function that computes the fitness value for an agent in which the agent consumes the path squares in a partial contiguous order. Recall that a full contiguous order means the next path square the agent consumes is immediately next to the previous path square consumed and that a partial contiguous order means the agent consumes segments of the path in a full contiguous order but consumes the path squares between segments in a non-contiguous order. Agents of successful runs consume the entire path in full contiguous order because the fitness function computes the agent's fitness with respect to partial contiguous order. Therefore, all agents of successful runs are contiguous agents. The average number of successful runs per path for the fifth experiment is 73.53 with a standard deviation of 41.27. 1,446 crossover paths for the fifth signature recognition program experiment have zero successful runs. The average number of unsuccessful runs per path for the fifth experiment is 26.47 with 0 to 100 as the range of unsuccessful runs per path for the fifth experiment. The average final generation number per successful run for the fifth experiment is 9.51 with 0 to 125 as the range of final generation number per successful run for the fifth experiment.

Unlike experiments one through four, the signature recognition program, for experiment five, writes the first highest fitness agent to the output file for an unsuccessful run of the signature recognition genetic algorithm. The agents of the unsuccessful runs of the fifth experiment have a strict partial contiguous fitness value; the agents consume only a portion of the path. Divide the fitness value of the unsuccessful run's agent by the length of the path to obtain the agent's partial contiguous fitness percentage, the percentage of the path's length consumed by the agent in a contiguous order. Compute the average partial contiguous fitness percentage across all of the agents for unsuccessful runs of the path. A count of the number of paths of each average partial contiguous fitness percentage range shows that the counts form a bell curve centered at the percentage range of 30% to 40%.

Experiments one through three, using the non-crossover paths as the data set, have very promising results. All 10,000 non-crossover paths produced at least one agent that consumes the path (signature) in a non-contiguous order. Also encouraging, only 20 of the 10,000 non-crossover paths did not have a contiguous agent amongst the agents found for the path. A visual examination of these 20 paths shows complex geometry, not in terms of path length or winding number but portions of the path coiled within the path. Experiments four and five, using the crossover paths as the data set, also have promising results. The crossover paths have a more complex geometry than the non-crossover paths. But, all 10,000 crossover paths in experiment four produced at least one agent that consumes the path (signature) in a non-contiguous order. The strict contiguous fitness function of experiment five produces results that are a step below the results of experiment 4 and the non-crossover path experiments one through three. Only 1,446

crossover paths for the fifth signature recognition program experiment have zero successful runs while the rest of the 10,000 crossover paths in experiment five produced at least one agent that consumes the path (signature) in a full contiguous order. 99% of the 3,799 crossover paths that have at least one unsuccessful run in experiment five have an average partial contiguous fitness percentage of 50% or more meaning the agent has the promise of recognizing more than half the path in a contiguous order. These experiments are just the beginning of the application of agents (finite state machines) to the recognition of handwritten signatures. Most signatures consist of more than one piece (path) with one or more breaks in the signature. Research can continue with one agent attempting to recognize all of the pieces of the signature or a separate agent handling recognition of its own piece of the signature. Finally, future data sets will be real signatures.

The optical character recognition problem is the second of the two problems addressed in this dissertation. The optical character recognition problem attempts to find a means to match a connected component in a page's image to a character in the character database. The Character Database Creation (CDC) program creates the database of characters in various font name and font size combinations. The Optical Character Recognition (OCR) program performs character recognition on pages of text using the character database created by the CDC program. Both programs have the same first four phases of code.

An ordinary flatbed scanner scans an 8.5 by 11 inch page of printed text with the resulting image stored in a file using the TIFF file format. Thresholding applied to the page's image file produces a strict binary black and white image of the scanned page of

text. White is the background color and black is the foreground color, the color of the text. The first phase finds all of the connected components in the image.

The second phase uses a genetic algorithm to determine the number of lines of text in the image. Each individual in the line recognition genetic algorithm's population is a line formed from the centroid points of two randomly chosen connected components in the image. An individual's fitness value is the number of lines parallel to the individual's line such that each connected component's centroid point is within a certain threshold distance from one of the fitness value lines. The distance threshold is 71% of the average height of the connected components in the image.

Eight non-skewed pages of text test the line recognition genetic algorithm. Five of the pages have text printed in portrait orientation with the text on the remaining three pages printed in landscape orientation. In addition, skew each page by four different angles to further test the line recognition genetic algorithm. The four angles are 6° counter-clockwise, 2° counter-clockwise, 2° clockwise and 6° clockwise. Therefore, the line recognition genetic algorithm tests a total of forty pages.

The line recognition genetic algorithm determined the correct number of lines for thirty-five of the forty test pages. The genetic algorithm recognized 2 lines for 1 actual line in the first text page skewed 6° counter-clockwise and the fifth text page skewed 2° clockwise and 6° clockwise. The genetic algorithm recognized a stray mark as a separate line in the sixth text page skewed 6° counter-clockwise and 2° counter-clockwise. Finally, the genetic algorithm determined the correct number of lines for the non-skewed eighth text page but the lines calculated by the genetic algorithm were skewed by 2° clockwise with respect to the text's actual lines.

The line recognition genetic algorithm provides the number of lines to the CDC/OCR program's third phase where the program arranges all of the connected components by line. The program merges neighboring components on a line, within a certain distance threshold of one another, into a single component. The fourth phase computes the 19 measure values for each connected component, the 7 Hu moment values, the 10 Flusser-Suk moment values, the axis-aligned rectangle area, and the minimum area rectangle area. After adding all of the characters of a font name and font size combination to the character database, the CDC program computes the minimum, maximum and average values for each measure for the font name and font size combination. These three values allow a visual comparison of the range of values of all font name and font size combinations for a particular measure.

An examination of the ranges for the axis aligned area and minimum area rectangle area measures reveals that within a particular font name the ranges increase as the font size increases. The Courier New font name has ranges that are smaller in size in relation to the other three font names. The ranges for each font name follow the same pattern and size making them little use for character recognition. Examining the ranges for the Hu moments shows that, for each font name and font size combination, the ranges become shorter as the moment number increases. The Arial font name ranges still have significant length for Hu moments 1 through 6. The one font name and font size combination that does stand out for the Hu moments is Times New Roman 12 point. The range for Times New Roman 12 point remains long for all 7 Hu moments while the ranges for all of the other font name and font size combinations shorten from Hu moment 1 to 7. A final interesting observation is the Courier New font name ranges suddenly

change from short to long between Hu moment 6 and 7. Examining the ranges for the Flusser-Suk moments reveals that the range for the Arial and Georgia font names become shorter as the moment number increases. The range for the Courier New and Times New Roman font names remain long for all ten of the Flusser-Suk moments.

The CDC/OCR program gives the line recognition genetic algorithm the orientation of the text in the page's image. In future work, the genetic algorithm can recognize the text orientation in addition to the lines on the page. An observation of the execution of the line recognition genetic algorithm reveals that it converges quickly, within the first 10 generations. The maximum number of possible individuals for the population is a computationally manageable size. Recall, an individual in the line recognition genetic algorithm is a pair of centroid points from two connected components in the page's image. The first text page contains 43 lines of text whose font name and size is Times New Roman 12 point. The text page has approximately 3,000 connected components. All possible pairings of connected components is 9,000,000 lines. Simple heuristics can eliminate a significant portion of lines leaving a small group of lines. Application of the line recognition genetic algorithm fitness function can quickly determine which line closely matches the lines of text in the page. Finally, future research can work on the moment based portion of the character recognition problem. Use of all 17 moments to recognize a connected component as one of the characters in the database yielded less than ideal results. Formulation of a genetic algorithm can determine the best set of moments to use in the recognition of each printable character. In addition, this research work can include other moments available in the literature.

Bibliography

[1] P. G. Anderson, C. Asbury, R. S. Gaborski and D. G. Tilley (1993), *Genetic Algorithm Selection of Features for Handwritten Character Identification*, The International Conference on Artificial Neural Networks Genetic Algorithms (ANNGA '93), Innsbruck, Austria, April 1993.

[2] D. Andre (1994), *Learning and Upgrading Rules for an OCR System Using Genetic Programming*, In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, Orlando, Florida, USA, IEEE Press, 1994.

[3] P. J. Angeline, D. B. Fogel and L. J. Fogel (1996), *A Comparison of Self-Adaptation Methods for Finite State Machines in a Dynamic Environment*, In: Evolutionary Programming V, L. J. Fogel, P. J. Angeline and T. Bäck (editors), MIT Press, Cambridge, MA, pp. 441-449, 1996.

[4] C. Aporntewan and P. Chongstitvatana (2000), *An On-line Evolvable Hardware for Learning Finite-State Machine*, In: Proceedings of International Conference on Intelligent Technologies, Bangkok, Thailand, pp.125-134, December 2000.

[5] J. W. Atmar (1976), *Speculation on the Evolution of Intelligence and Its Possible Realization in Machine Form*, Ph.D. Dissertation, New Mexico State University, 1976.

[6] T. Bäck, G. Rudolph and H. P. Schwefel (1993), *Evolutionary Programming and Evolution Strategies: Similarities and Differences*, In: Proceedings of the Second Annual Conference on Evolutionary Programming, La Jolla, CA, USA, Evolutionary Programming Society, pp. 11-22, 1993.

[7] A. Berlanga, J. Garcia-Herrero, J.M. Molina, J. Besada and J. Portillo (2002), *OCR Parameters Tuning by Means of Evolution Strategies for Aircraft's Tail Number Recognition*, In: Proceedings of the 2002 Congress on Evolutionary Computation (CEC '02), vol. 1, pp. 902-907, 2002.

[8] M. Boryczka (2005), *Eliminating Introns in Ant Colony Programming*, In: Fundamenta. Informaticae, vol. 68, no. 1-2, pp. 1-19, 2005.

[9] H. J. Bremermann (1958), *The Evolution of Intelligence. The Nervous System as a Model of Its Environment*, Technical report, no. 1, contract no. 477(17), Department of Mathematics, University of Washington, Seattle, July 1958.

[10] M. E. Celebi and Y. A. Aslandogan (2005), *A Comparative Study of Three Moment-Based Shape Descriptors*, In: International Conference on Information Technology: Coding and Computing 2005 (ITCC 2005), vol. 1, pp. 788-793, April 2005.

[11] S. H. Cha and C. C. Tappert (2003), *Optimizing Binary Feature Vector Similarity Measure Using Genetic Algorithm*, In: Proceedings of the Seventh International Conference on Document Analysis and Recognition (ICDAR ’03), Edinburgh, Scotland, 2003.

[12] F. Dellaert and J. Vandewalle (1994), *Automatic Design of Cellular Neural Networks by Means of Genetic Algorithms: Finding a Feature Detector*, In: Proceedings of the Fourth International IEEE Workshop on Cellular Neural Networks and Their Applications (CNNA '94), 1994.

[13] C. De Stefano, A. Della Cioppa and A. Marcelli (2002), *Character Preclassification Based on Genetic Programming*, In: Pattern Recognition Letters, vol. 23, no. 12, pp. 1439-1448, October 2002.

[14] D. Eberly (2003), *Minimum Area Rectangle Containing a Convex Polygon*, Geometric Tools, Inc., 2003.

[15] M. Ebner (1998), *On the Evolution of Interest Operators Using Genetic Programming*, In: Late Breaking Papers at EuroGP '98: the First European Workshop on Genetic Programming, R. Poli, W. B. Langdon, M. Schoenauer, T. Fogarty and W. Banzhaf (editors), The University of Birmingham, U.K., pp. 6-10, 1998.

[16] J. Flusser and T. Suk (1993), *Pattern Recognition by Affine Moment Invariants*, In: Pattern Recognition, vol. 26, no. 1, pp. 167-174, 1993.

[17] J. Flusser and T. Suk (1994), *Affine Moment Invariants: A New Tool for Character Recognition*, In: Pattern Recognition Letters, vol. 15, pp. 433-436, 1994.

[18] J. Flusser and T. Suk (2004), *Graph Method for Generating Affine Moment Invariants*, In: Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004), J. Kittler (editor), IEEE Computer Society, Los Alamitos, pp. 192-195, 2004.

[19] D. B. Fogel (1993), *Evolving Behaviors in the Iterated Prisoner's Dilemma*, In: Evolutionary Computation, vol. 1, no. 1, pp. 77-97, 1993.

[20] D. B. Fogel and L. J. Fogel (1996a), *Using Evolutionary Programming to Schedule Tasks on a Suite of Heterogeneous Computers*, In: Computers & Operations Research, vol. 23, no. 6, pp. 527-534, 1996.

[21] D. B. Fogel and L. J. Fogel (1996b), *Preliminary Experiments on Discriminating between Chaotic Signals and Noise Using Evolutionary Programming*, In: Genetic Programming 1996: Proceedings of the First Annual Conference, J. R. Koza, D. E. Goldberg, D. B. Fogel and R. L. Riolo (editors), MIT Press, Cambridge, MA, pp. 512-520, 1996.

[22] D. B. Fogel and L. J. Fogel (1996c), *An Introduction to Evolutionary Programming*, In: Proceedings of Evolution Artificielle 95, Springer-Verlag, Berlin, pp. 21-33, 1996.

[23] D. B. Fogel, L. J. Fogel, and J. W. Atmar (1991), *Meta-Evolutionary Programming*, In: Proceedings of the 25th Asilomar Conference on Signals, Systems & Computers, R. R. Chen (editor), Pacific Grove, CA, pp. 540-545, 1991.

[24] D. B. Fogel, L. J. Fogel, W. Atmar and G.B. Fogel (1992), *Hierarchic Methods of Evolutionary Programming*, In: Proceedings of the First Annual Conference on Evolutionary Programming, D. B. Fogel and W. Atmar (editors), La Jolla, CA, Evolutionary Programming Society, pp. 175-182, 1992.

[25] D. B. Fogel, L. J. Fogel, and V. W. Porto (1990), *Evolving Neural Networks*, In: Biological Cybernetics, vol. 63, no. 6, pp. 487-493, 1990.

[26] L. J. Fogel, P. J. Angeline and D. B. Fogel (1994), *A Preliminary Investigation on Extending Evolutionary Programming to Include Self-Adaptation on Finite State Machines*, In: Informatica, vol. 18, no. 4, pp. 387-398, 1994.

[27] L. J. Fogel, P. J. Angeline and D. B. Fogel (1995), *An Evolutionary Programming Approach to Self-Adaptation in Finite State Machines*, In: Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming, J. R. McDonnell, R. G. Reynolds and D. B. Fogel (editors), MIT Press, Cambridge, MA, pp. 355-365, 1995.

[28] L. J. Fogel, D. B. Fogel and W. Atmar (1993), *Evolutionary Programming for ASAT Battle Management*, In: Proceedings of the 27th Asilomar Conference on Signals, Systems and Computers, A. Singh (editor), IEEE Computer Society Press, Los Alamitos, CA, pp. 617-621, 1993.

[29] L. J. Fogel, A. J. Owens and M. J. Walsh (1966), *Artificial Intelligence through Simulated Evolution*, John Wiley, New York, 1966.

[30] A. S. Fraser (1957), *Simulation of Genetic Systems by Automatic Digital Computers. I. Introduction*, Australian Journal of Biological Sciences, vol. 10, pp. 484-491, 1957.

[31] A. Guarda, C. Le Gal and A. Lux (1998), *Evolving Visual Features and Detectors*, In: International Symposium on Computer Graphics, Image Processing, and Vision, Rio de Janeiro, Brazil, 1998.

[32] S. Hengpraprohm and P. Chongstitvatana (2001), *Selective Crossover in Genetic Programming*, In: International Symposium on Communications and Information Technology, Thailand, pp. 534-537, November 2001.

[33] T. Hikage, H. Hemmi and K. Shimohara (1997), *Progressive Evolution Model Using a Hardware Evolution System*, In: The Second International Symposium on Artificial Life and Robotics (AROB '97), pp. 18-21, 1997.

[34] J. Holland (1975), *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI, 1975.

[35] M. K. Hu (1962), *Visual Pattern Recognition by Moment Invariants*, In: IRE Transactions on Information Theory, vol. 8, pp. 179-187, 1962.

[36] H. Iba and M. Terao (2000), *Controlling Effective Introns for Multi-Agent Learning by Genetic Programming*, In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), Las Vegas, Nevada, USA, pp. 419-426, July 2000.

[37] N. Kharma, T Kowaliw, E. Clement, C. Jensen, A. Youssef and Jie Yao (2004), *Project CellNet: Evolving an Autonomous Pattern Recognizer*, In: International Journal of Pattern Recognition and Artificial Intelligence, vol. 18, no. 6, pp. 1039-1056, 2004.

[38] T. Kowaliw, N. Kharma, C. Jensen, H. Mognieh and J. Yao (2005), *Using Competitive Co-Evolution to Evolve Better Pattern Recognizers*, In: The International Journal of Computational Intelligence and Applications, vol. 5, no. 3, pp. 305-320, 2005.

[39] J. R. Koza (1993), *Simultaneous Discovery of Reusable Detectors and Subroutines Using Genetic Programming*, In: Proceedings of the Fifth International Conference on Genetic Algorithms, S. Forrest (editor), Morgan Kaufmann Publishers Inc., San Mateo, CA, pp. 295-302, 1993.

[40] S. R. Ladd (1995), *Genetic Algorithms in C++*, M&T Books, 1995.

[41] W. B. Langdon (1998), *Better Trained Ants*, In: Late Breaking Papers at the First European Workshop on Genetic Programming, R. Poli, W. B. Langdon, W. Banzhaf, M. Schoenauer and T. C. Fogarty (editors), Paris, pp. 11-13, April 1998.

[42] P. L'Ecuyer (1988), *Efficient and portable combined random number generators*, In: Communications of the ACM, vol. 31, pp. 742-774, June 1988.

[43] L. Lin, X. Wang and D. Yeung (2005), *Combining Multiple Classifiers Based on a Statistical Method for Handwritten Chinese Character Recognition*, In: International Journal of Pattern Recognition and Artificial Intelligence, vol. 19, no. 1, pp. 1027-1040, 2005.

[44] J. Liu, D. A. Maluf and Y. Y. Tang (1997), *Toward Evolutionary Autonomous Agents for Computational Perception*, In: Proceedings of the 1997 IEEE International Conference on System, Man, and Cybernetics (SMC '97), vol. 4, pp. 3096-3101, October 1997.

[45] J. Liu and Y. Y. Tang (1999), *Adaptive Image Segmentation with Distributed Behavior-Based Agents*, In: IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 21, no. 6, pp. 544-551, June 1999.

[46] J. D. Lohn and J. A. Reggia (1995), *Discovery of Self-replicating Structures Using a Genetic Algorithm*, In: Proceedings of the 1995 IEEE International Conference on Evolutionary Computation (ICEC'95), IEEE, Piscataway, New Jersey, pp.678-683, 1995.

[47] S. Luke (2000), *Code Growth Is Not Caused by Introns*, In: GECCO-2000: Late Breaking Papers, Las Vegas, Nevada, USA, pp. 228-235, July 2000.

[48] S. Luke (2003), *Modification Point Depth and Genome Growth in Genetic Programming*, In: Evolutionary Computation, vol. 11, no. 1, pp. 67-106, Spring 2003.

[49] J. McDonnell, D. B. Fogel, L. J. Fogel, C. Rindt and W. Recker (1995), *Evolving Optimal Ramp Control Rules*, In: International Journal of Expert Systems, vol. 8, no. 3, pp. 287-308, 1995.

[50] J. R. McDonnell, W. C. Page, D. B. Fogel, and L. J. Fogel (1997), *Optimizing Fuel Distribution through Evolutionary Programming*, In: Evolutionary Programming VI: Proceedings of the Sixth Annual Conference on Evolutionary Programming, P. J. Angeline, R. G. Reynolds, J. R. McDonnell, and R. C. Eberhart (editors), Springer, Berlin, pp. 373-381, 1997.

[51] N. F. McPhee and J. D. Miller (1995), *Accurate Replication in Genetic Programming*, In: Proceedings of the Sixth International Conference on Genetic Algorithms, L. J. Eshelman (editor), Morgan Kaufmann, pp. 303-309, 1995.

[52] J. Miller (2001), *What Bloat? Cartesian Genetic Programming on Boolean Problems*, In: GECCO-2001: Late Breaking Papers, pp. 295–302, July 2001.

[53] M. Mitchell and S. Forrest (1994), *Genetic Algorithms and Artificial Life*, In: Artificial Life, vol. 1, no. 3, pp. 267-289, 1994.

[54] P. Nordin, W. Banzhaf and F. D. Francone (1997), *Introns in Nature and in Simulated Structure Evolution*, In: Bio-Computation and Emergent Computation, D. Lundh, B. Olsson and A. Narayanan (editors), World Scientific Publishing, September 1997.

[55] P. Nordin, W. Banzhaf and F. D. Francone (1999), *Compression of Effective Size in Genetic Programming*, In: Foundations of Genetic Programming, T. Haynes, W. B. Langdon, U. O'Reilly, R. Poli and J. Rosca (editors), Orlando, Florida, USA, pp. 57-60, 1999.

[56] L. S. Oliveira, N. Benahmed, R. Sabourin, F. Bortolozzi and C. Y. Suen (2001), *Feature Subset Selection Using Genetic Algorithms for Handwritten Digit Recognition*, In: Proceedings of the 14th Brazilian Symposium on Computer Graphics and Image Processing, Florianopolis, Brazil, IEEE Computer Society, pp. 362–369, 2001.

[57] X. Pan, X. Ye and S. Zhang (2005), *A Hybrid Method for Robust Car Plate Character Recognition*, In: Engineering Applications of Artificial Intelligence, vol. 18, pp. 963–972, 2005.

[58] V. W. Porto, D. B. Fogel, and L. J. Fogel (1995), *Alternative Neural Network Training Methods*, In: IEEE Expert, vol. 10, no.3, pp. 16-22, June 1995.

[59] V. W. Porto, D. B. Fogel and L. J. Fogel (1998), *Generating Novel Tactics through Evolutionary Computation*, In: SigArt Bulletin, ACM Press, pp. 8-14, Fall 1998.

[60] V. W. Porto, D. B. Fogel, L. J. Fogel, G. B. Fogel, N. Johnson and M. Cheung (2005), *Classifying Sonar Returns for the Presence of Mines: Evolving Neural Networks and Evolving Rules*, In: 2005 IEEE Symposium on Computational Intelligence for Homeland Security and Personal Safety, D. B. Fogel and V. Piuri (editors), IEEE Press, Piscataway, NJ, pp. 123-130, 2005.

[61] V. W. Porto, L. J. Fogel and D. B. Fogel (2004), *Using Evolutionary Computation for Seismic Signal Processing: A Homeland Security Application*, In: Proceedings of 2004 IEEE International Conference on Computational Intelligence for Homeland Security and Personal Safety, Venice, Italy, IEEE Press, pp. 62-66, 2004.

[62] P. W. H. Smith and K. Harries (1999), *Code Growth, Explicitly Defined Introns, and Alternative Selection Schemes*, In: Evolutionary Computation, vol. 6, no. 4, pp. 339-360, 1999.

[63] H. Sossa and J. Flusser (2004), *Refined Method for the Fast and Exact Computation of Moment Invariants*, In: Progress in Pattern Recognition, Image Analysis, and Applications, A. Sanfeliu (editor), Lecture Notes on Computer Science 3287, Springer, Berlin, pp. 487-494, 2004.

[64] T. Soule and J. A. Foster (1997), *Code Size and Depth Flows in Genetic Programming*, In: Genetic Programming 1997: Proceedings of the Second International Conference, J. R. Koza et al. (editors), Morgan Kaufmann, pp. 313-320, 1997.

[65] S. Sural and P. K. Das (2001), *A Genetic Algorithm for Feature Selection in a Neuro-Fuzzy OCR System*, In: Proceedings of the 6th International Conference on Document Analysis and Recognition (ICDAR'01), pp. 987–991, 2001.

[66] K. Thearling (1992), *Putting Artificial Life to Work*, In: Parallel Problem Solving from Nature 2, R. Männer and B. Manderic (editors), North-Holland: Amsterdam, September 1992.

[67] B. C. Wallet, D. J. Marchette, J. L. Solka (1996), *A Matrix Representation for Genetic Algorithms*, In: Automatic object recognition VI, The International Society for Optical Engineering, volume SPIE-2756, pp. 206-214, April 1996.

[68] F. M. Waltz (1994a), *SKIPSM: Separated-Kernel Image Processing Using Finite-State Machines*, Paper No. 36, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[69] F. M. Waltz (1994b), *Application of SKIPSM to Binary Template Matching*, Paper No. 39, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[70] F. M. Waltz (1994c), *Application of SKIPSM to Grey-level Morphology*, Paper No. 40, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[71] F. M. Waltz (1994d), *Application of SKIPSM to the Pipelining of Certain Global Image Processing Operations*, Paper No. 41, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[72] F. M. Waltz and H. H. Garnaoui (1994a), *Application of SKIPSM to Binary Morphology*, Paper No. 37, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[73] F. M. Waltz and H. H. Garnaoui (1994b), *Fast Computation of the Grassfire Transform Using SKIPSM*, Paper No. 38, In: Proceedings of the SPIE Conference on Machine Vision Applications, Architectures, and Systems Integration III, Boston, SPIE vol. 2347, November 1994.

[74] X. Wei, S. Ma and Y. Jin (2005), *Segmentation of Connected Chinese Characters Based on Genetic Algorithm*, In: Proceedings of the 2005 Eight International Conference on Document Analysis and Recognition (ICDAR'05), pp. 645–649, 2005.