

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A

The Effect of the Dynamic Reorganization
of a Finite State Machine Genome
on the Efficiency of a Genetic Algorithm

by

Natalie Hammerman

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

1999

UMI Number: 9917658

**Copyright 1999 by
Hammerman, Natalie**

All rights reserved.

**UMI Microform 9917658
Copyright 1999, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© 1999

NATALIE HAMMERMAN

All Rights Reserved

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

1/11/99
Date

Robert R. Goldberg
Chair of Examining Committee

1/11/99
Date

Jerry Waxman
Executive Officer

Dr. Robert Goldberg, chair (Queens College)
Dr. Michael Anshel (City College)
Dr. David Fogel (Natural Selection, Inc.)
Dr. Jacob Shapiro (Baruch)
Dr. Jerry Waxman (Queens College)
Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract**The Effect of the Dynamic Reorganization of a Finite State Machine Genome on the Efficiency of a Genetic Algorithm**

by

Natalie Hammerman**Advisor: Professor Robert Goldberg**

Strategies for solving different types of problems can be represented as a finite state machine (FSM). In artificial life and artificial intelligence research, such problems use finite state machines as the genotype (operand) for genetic algorithms (GAs). Strategies which are FSM-specific and designed to improve the convergence of the genetic algorithm for FSM genomes are investigated. Because a single finite state machine has different representations (simply by changing state names), two reorganization operators (named SFS and MTF) were developed so that identical machines would appear the same and not have to compete for their share of the next generation. The operators were designed with the intent of enhancing schemata growth for an FSM genome by reorganizing a population of these machines during run time. Experiments were performed with these new operators in order to determine how they would affect the efficiency of genetic algorithms. Strategies were then developed to permit evaluation of the data. It was found that MTF outperforms the other methods when the average size of solutions is less than 45% of the genome size. Based on this evaluation, it is deemed that the MTF operator be incorporated into a GA application.

Preface

"Basic research is where scientists are free to pursue their curiosity and interrogate Nature, not with any short-term practical end in view but to seek knowledge for its own sake." (Sagan 1995, p. 14)

George Boole's *The Laws of Thought*, featuring his now famous and much used "Boolean logic," appeared (1854) before a technology adequate for construction of logic machines was in place. But, when that (electronic) technology did become available, computers and their programming languages sprang up almost overnight, because pioneers like Wiener, Turing, Simon, and Burks were steeped in the "pure" researches that had been made with the Boolean and post-Boolean symbolic logics. To men quite accustomed to formal systems, and the rigorous modes of thought that produce and manipulate them, the construction and use of computer technology at hardware and software levels came as second nature.

We are . . . in that "basic research" period, inventing and trying out various tools of thought. . . . hard as this is for the result-oriented pragmatist to understand— . . . the most general insights we can extract from our present efforts are obtainable from focussing on *how* we establish our problem space, set and work toward our goals, measure our progress and learn from our failures, rather than what is produced. . . . The next generation . . . *will* build on the insights disclosed in the *approaches* we have taken. . . .

They, in sum, will advance and eventually embody the concepts and modes absorbed from our present-day, seemingly esoteric exercises, . . .

(Morris 1989, p. 388)

Reprinted with permission of Addison Wesley Longman.

Acknowledgements

There is neither enough time nor space to thank all the people who have helped me move from a frightened graduate student to the recipient of a PhD, so I have singled out the few most responsible for that transformation.

I want to thank *all* the members of my committee for helping me through my research. Dr. Robert Goldberg has my sincerest appreciation not only for being my advisor, but also for providing valuable guidance by treating me as a colleague rather than as an inexperienced subordinate. In addition, special thanks go to Dr. David Fogel for agreeing to serve on the committee of a complete stranger who e-mailed him requesting his assistance.

Both Drs. S.A. Ghozati and Robert Goldberg have acted as mentors to me since I returned to school for graduate studies. While it was Dr. Goldberg who guided me through the thesis, Dr. Ghozati was instrumental in getting me to the point where I could begin to approach it. Both educators encouraged me to use the knowledge I gained in class as a starting point. They provoked me to question and challenge that material, and use it as a base for new ideas. They both invited me to study next to them rather than under them, encouraged me to move beyond the limitations of the knowledge I gained, and most importantly taught me that my ability to learn was not limited to the classroom and books.

My husband encouraged me to return to school 16 years after I earned my bachelor's, had unfailing belief in me, and encouraged me to go on when I doubted myself. He has made many sacrifices throughout the years to permit me to continue my education.

I offer sincerest thanks not only to these people, but for all who have helped me move towards this degree.

Most of chapters 1, 2, 4, 5, and 6, and appendix A are under copyright held by CRC Press, LLC. These chapters are reprinted under an agreement with them.

Table of Contents

	page
1. Introduction and background	1
1.1 Examples of phenotypes	1
1.2 Finite state machines as the phenotype	6
2. The genetic algorithm	10
2.1 A simple genetic algorithm	10
2.2 Variations on the simple genetic algorithm	19
2.2.1 Variations on fitness	20
2.2.2 Variations on the selection process	22
2.2.3 Crossover and mutation variations, and other operators	24
2.2.4 Schema: definition and theory	26
2.2.5 Schemata at work: an example	30
2.2.6 Fitness landscape	32
2.2.7 One strategy to reduce the chance of premature convergence	35
2.2.8 Variations on termination	39
2.3 Reflections on the genetic algorithm	40
3. Project aims and considerations	42
4. The benchmark	47
4.1 The finite state machine as a genome	47
4.2 The benchmark: description and theory	49
4.3 The coding	62
5. The new operators	92
5.1 The SFS algorithm	93
5.1.1 SFS: description and coding	93
5.1.2 Examples of SFS reorganization	104
5.2 The MTF algorithm	107
5.2.1 MTF: description and coding	108
5.2.2 Example of an MTF reorganization	110
5.3 Competition	111
6. Interesting ants bred during experimentation	126
6.1 Best observed time: 181 time steps	128
6.2 Intermediate FSMs	132
6.3 8-state perfect scoring FSMs	137

Table of Contents Continued

	page
7. Evaluation criteria	143
7.1 The context	144
7.2 Evaluating the evolutionary process	147
7.3 Evaluating convergence	150
7.3.1 Evaluation factors	151
7.3.2 Considerations in the calculation of $G\mu$ and $T\mu$	153
7.3.3 Formulas for the summation of the required infinite sequences	154
7.3.4 Definitions of variables	155
7.3.5 Calculating $G\mu$, the mean number of generations	157
7.3.6 Calculating the variance	160
7.3.7 Calculation of mean processing time and its variance	164
7.3.8 Use of $G\mu$ and $T\mu$	165
7.3.9 $G\mu$ and $T\mu$ formulas for a sample	167
7.3.10 Ranking criteria	171
7.3.10.1 The ranking procedures implemented	171
7.3.10.2 Code for probability of getting a solution first	176
7.3.10.3 Summary of ranking criteria	185
8. Evaluation of the algorithms	188
8.1 Evaluation of the evolutionary process	188
8.1.1 The experimentation	189
8.1.2 Data Analysis	190
8.1.2.1 B1, MTF1, and SFS1 compared	191
8.1.2.1.1 64K population	191
8.1.2.1.2 16K population	193
8.1.2.2 B and SFS compared with/out competition	195
8.1.2.3 B and MTF compared with/out competition	199
8.1.2.4 Periodic reorganization	200
8.2 Evaluation of efficiency	208
8.2.1 The experimentation	208
8.2.2 Evaluation of the individual experiments	210
8.2.2.1 Population size 16K; 30 seeds	211
8.2.2.1.1 Failure rate and K	211
8.2.2.1.2 Probability of fewer generations or less processor time	212
8.2.2.1.3 Grouping of best five runs	215
8.2.2.1.4 G_{AV} and $T_{AV:S}$	217
8.2.2.1.5 $G\mu$ and $T\mu$	219
8.2.2.1.6 Machine size	221
8.2.2.1.7 Summary of experiment results	224

Table of Contents Continued

	page
8.2.2.2 Population size 32K; 30 seeds . . .	225
8.2.2.2.1 Failure rate and K . . .	225
8.2.2.2.2 Probability of fewer generations .	226
8.2.2.2.3 Grouping of best five runs based on number of generations . . .	227
8.2.2.2.4 G_{AV} . . .	228
8.2.2.2.5 G_{μ} . . .	229
8.2.2.2.6 Machine size . . .	230
8.2.2.2.7 Summary of experiment results .	233
8.2.2.3 Population size 64K; 30 seeds . . .	234
8.2.2.3.1 Failure rate and K . . .	234
8.2.2.3.2 Probability of fewer generations .	235
8.2.2.3.3 Grouping of best five runs based on number of generations . . .	236
8.2.2.3.4 G_{AV} . . .	237
8.2.2.3.5 G_{μ} . . .	238
8.2.2.3.6 Machine size . . .	239
8.2.2.3.7 Summary of experiment results .	243
8.2.2.4 Population size 16K; 100 seeds . . .	244
8.2.2.4.1 Failure rate and K, and probability of fewer generations . . .	244
8.2.2.4.2 Grouping of best five runs based on number of generations . . .	246
8.2.2.4.3 G_{AV} . . .	246
8.2.2.4.4 G_{μ} . . .	247
8.2.2.4.5 Machine size . . .	248
8.2.2.4.6 Summary of experiment results .	250
8.2.2.5 Population size 32K; 100 seeds . . .	250
8.2.2.5.1 Failure rate and K . . .	251
8.2.2.5.2 Probability of fewer generations .	251
8.2.2.5.3 Grouping of best five runs based on number of generations . . .	252
8.2.2.5.4 G_{AV} . . .	253
8.2.2.5.5 G_{μ} . . .	254
8.2.2.5.6 Machine size . . .	255
8.2.2.5.7 Summary of experiment results .	257
8.2.2.6 Population size 64K; 100 seeds . . .	257
8.2.2.6.1 Failure rate and K . . .	258
8.2.2.6.2 Probability of fewer generations .	258
8.2.2.6.3 Grouping of best five runs based on number of generations . . .	259
8.2.2.6.4 G_{AV} . . .	260
8.2.2.6.5 G_{μ} . . .	261

Table of Contents Continued

	page
8.2.2.6.6 Machine size	262
8.2.2.6.7 Summary of experiment results	264
8.2.3 Evaluation of individual criterion	264
8.2.3.1 G_{AV}	266
8.2.3.2 Failure rate and K	268
8.2.3.3 Probability of fewer generations	269
8.2.3.4 The $G_{\mu s}$	271
8.3 Evaluation summary	274
9. Performance across a new problem set	279
9.1 The experimentation	280
9.1.1 The languages	280
9.1.2 Algorithmic and programming considerations	284
9.2 Evaluation criteria	293
9.3 Evaluation	297
9.4 Performance of MTF	300
10. Conclusion	302
10.1 Summary of evaluation criteria	303
10.2 Conclusions	308
10.3 Future research	311
Appendix A: SFS walk-through	314
Appendix B: Proofs for sums of infinite series	339
Appendix C: Graphs comparing individual runs for the trail problem	344
Appendix D: Worksheets for language recognition problem	434
References	470

Tables

	page
2.1 Initial Population	13
2.2 First Generation after Selection	15
2.3 First Generation after Crossover	16
2.4 First Generation after Mutation	17
4.1 Four State FSM	58
5.1 FSM of Table 4.1 after Step 1 of SFS	105
5.2 FSM of Table 4.1 after Next States for q0 Reassigned	105
5.3 FSM of Table 4.1 after SFS	106
5.4 FSM of Table 4.1 after MTF	107
5.5 Template for a Family of Ants with Differences in One Transition/Output	114
5.6 Template for a Few Perfect Scoring Ants with Differences in Two Transitions/Outputs	115
5.7 Comparison of Runs with and without Competition	118-119
6.1 Template for 12-state Perfect Scoring Ants	129
6.2 Substitutions for ### and @@@ in Table 6.1	129
9.1 Parameters for the Languages	292
A.1 FSM before SFS	315
A.2-A.20 contains intermediate steps as SFS is applied to FSM of table A.1	316-336
A.21 Correlation of States between the FSMs in Tables A.1 and A.22	337
A.22 FSM of Table A.1 after SFS	338

Figures		page
1.1	Equivalent FSMs	7
2.1	Sample Run	18
2.2	Comparison of runs with different phenotypes.	32
2.3	a) Two-dimensional square b) Three-dimensional cube c) Four-dimensional hypercube with top-to-bottom and side-to-side adjacency.	33
2.4	Fitness Landscape for a four-bit genome with phenotype x =a binary integer and fitness $f(x)=x+1$	34
2.5	Fitness Landscape for a four-bit genotype with two different phenotypes.	34
4.1	The John Muir Tail	50
4.2	The Santa Fe Trail	51
4.3	The John Muir Trail as it appears to the ant	52
4.4	Strategy favoring right turns when the next step is unmarked	54
4.5	Strategy alternates left and right turns when next step is unmarked.	55
4.6	32-state/453-bit FSM Genome Map	57
4.7	Calculate 2^n	65-66
4.8	Bitstring to Bytestring Conversion	67
4.9	Bytestring to Bitstring Conversion	67
4.10	Binary to Decimal Conversion	68
4.11	Decimal to Binary Conversion	69
4.12	Bytestring to Table Conversion	70
4.13	Table to Bytestring Conversion	71
4.14	Determine an Ant's Fitness	72-75

Figures continued

	page
4.15 Output Successful FSMs	77-78
4.16 The main Part of the GA	78
4.17 Check for Contradictory Constant Values	79-80
4.18 Print Headings on Output Files	80-81
4.19 Close Output Files	81
4.20 Initialization	82
4.21 Calculating and Outputting the Statistics for Each Generation	83-85
4.22 Breeding the Next Generation	85
4.23 Mating	87
4.24 Mutation	87-88
4.25 Listing of File with st Extension from Sample Run	89-90
4.26 Listing of File with ps Extension from Sample Run	90
5.1 Interchange Rows of a Table	94
5.2 Interchange States Based on Name Change	94
5.3 Reorganize Parent Pool	95
5.4 SFS Function Assigning Next State for a Given Present State/Input	97
5.5 The Second Step of the SFS Algorithm	98-99
5.6 The SFS Algorithm	101-103
5.7 The MTF Algorithm	109
5.8 Example of Premature Convergence	112-113
5.9 Example of an Unsuccessful Run	116-117
5.10 Comparison of Runs with and without Competition	118

Figures continued

5.11 Competition Procedure	page 120
5.12 The Procedure to Calculate and Output the Statistics for Each Generation: Modified for Competition	123-124
6.1 The John Muir Trail with Marked and Unmarked Trail Steps Labelled	127
6.2 12-state Perfect Scoring Ant with Time=181	130
6.3 11-state Perfect Scoring Ant with Time=188	132
6.4 10-state Perfect Scoring Ant with Time=185	134
6.5 9-state Perfect Scoring Ant with Time=187	135
6.6 8-state Perfect Scoring Ant with Time=193	137
6.7 8-state Perfect Scoring Ant with Time=193 Bred by a Benchmark Run	139
6.8 FSM of Figure 6.7 after MTF	139
6.9 8-state Perfect Scoring Ant with Time=199	141
7.1 Example of a 30-Seed Run	145-146
7.2 Ranking Based on f Failures Followed by a Successful Run	176-178
7.3 Calculation of Percent Increase	183
8.1 Comparison of B1, SFS1, and MTF1. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	192
8.2 Comparison of B1, SFS1, and MTF1. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	194
8.3 Comparison of B1, B2, SFS1, and SFS2. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	196
8.4 Comparison of B1, B2, SFS1, and SFS2. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	197
8.5 Comparison of B1, B2, MTF1, and MTF2. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	198

Figures continued

	page
8.6 Comparison of B1, B2, MTF1, and MTF2. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	199
8.7 Comparison of MTF1 runs with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	202
8.8 Comparison of MTF1 runs with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	203
8.9 Comparison of MTF2 runs with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	204
8.10 Comparison of MTF2 runs with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	205
8.11 Comparison of MTF1- i and MTF2- i with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 64K.	206
8.12 Comparison of MTF1- i and MTF2- i with MTF applied every i^{th} generation, $i \in \{1,2,3,4,5\}$. Averages over 30 runs of maximum fitness of each generation. Population size 16K.	207
8.13 Population size=16K; 30 seeds. Ranked by number of failures.	211
8.14 Population size=16K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	213
8.15 Population size=16K; 30 seeds. Ranked using the probability that a method will get a solution first; based on processor time for a successful run combined with the processor time needed for failed runs.	214
8.16 Population size=16K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs.	216
8.17 Population size=16K; 30 seeds. Ranked by the processor time taken to find a solution—grouped runs.	216

Figures continued

	page
8.18 Population size=16K; 30 seeds. Ranked by G_{AV} .	218
8.19 Population size=16K; 30 seeds. Ranked by $T_{AV:S}$.	218
8.20 Population size=16K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population.	219
8.21 Population size=16K; 30 seeds. Ranked by $T\mu$ calculated using the formula for the population.	220
8.22 Population size=16K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	220
8.23 Population size=16K; 30 seeds. Ranked by $T\mu$ calculated using the formula for the sample.	220-221
8.24 Population size=16K; 30 seeds. Ranked by machine size—grouped runs.	222-223
8.25 Population size=16K; 30 seeds. Ranked by average machine size.	223
8.26 Population size=32K; 30 seeds. Ranked by number of failures.	226
8.27 Population size=32K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	226
8.28 Population size=32K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs.	227
8.29 Population size=32K; 30 seeds. Ranked by G_{AV} .	228
8.30 Population size=32K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population.	229
8.31 Population size=32K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	229
8.32 Population size=32K; 30 seeds. Ranked by machine size—grouped runs.	231-232
8.33 Population size=32K; 30 seeds. Ranked by average machine size.	233

Figures continued

	page
8.34: Population size=64K; 30 seeds. Ranked by number of failures.	234
8.35: Population size=64K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	235
8.36: Population size=64K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs.	236
8.37: Population size=64K; 30 seeds. Ranked by G_{AV} .	237
8.38: Population size=64K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population.	238
8.39: Population size=64K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	239
8.40: Population size=64K; 30 seeds. Ranked by machine size—grouped runs.	240-242
8.41: Population size=64K; 30 seeds. Ranked by average machine size.	243
8.42: Population size=16K; 100 seeds. Ranked by number of failures.	245
8.43: Population size=16K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	245
8.44: Population size=16K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs.	246
8.45: Population size=16K; 100 seeds. Ranked by G_{AV} .	247
8.46: Population size=16K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population.	248
8.47: Population size=16K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	248
8.48: Population size=16K; 100 seeds. Ranked by machine size—grouped runs.	249
8.49: Population size=16K; 100 seeds. Ranked by average machine size.	250
8.50: Population size=32K; 100 seeds. Ranked by number of failures.	251

Figures continued

	page
8.51: Population size=32K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	252
8.52: Population size=32K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs.	253
8.53: Population size=32K; 100 seeds. Ranked by G_{AV} .	253
8.54: Population size=32K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population.	254
8.55: Population size=32K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	255
8.56: Population size=32K; 100 seeds. Ranked by machine size—grouped runs.	256
8.57: Population size=32K; 100 seeds. Ranked by average machine size.	256-257
8.58: Population size=64K; 100 seeds. Ranked by number of failures.	258
8.59: Population size=64K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs.	259
8.60: Population size=64K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs.	260
8.61: Population size=64K; 100 seeds. Ranked by G_{AV} .	260
8.62: Population size=64K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population.	261
8.63: Population size=64K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample.	262
8.64: Population size=64K; 100 seeds. Ranked by machine size—grouped runs.	263
8.65: Population size=64K; 100 seeds. Ranked by average machine size.	264
8.66 Summary for ranking by G_{AV} , with speedup over B1.	267

Figures continued

	page
8.67 Summary for ranking by number of failures and K, with speedup over B1 based on K.	268
8.68 Summary for ranking for the probability that a method will get a solution first, with percent increase over B1.	270
8.69 Summary for ranking by $G\mu$ using the formula for the population, with speedup over B1.	272-273
8.70 Summary for ranking by $G\mu$ using the formula for the sample, with speedup over B1.	273-274
9.1 16-state/148-bit FSA Genome Map	290
C.1a: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64K. Identical initial population is basis for comparison.	344-348
C.1b: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64K. Evolutionary progress is basis for comparison.	349-353
C.1c: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64K. Location of optimal individual is basis for comparison.	354-358
C.2a: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16K. Identical initial population is basis for comparison.	359-363
C.2b: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16K. Evolutionary progress is basis for comparison.	364-368
C.2c: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16K. Location of optimal individual is basis for comparison.	369-373

Figures continued

	page
C.3a: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 64K. Identical initial population is basis for comparison.	374-378
C.3b: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 64K. Evolutionary progress is basis for comparison.	379-383
C.3c: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 64K. Location of optimal individual is basis for comparison.	384-388
C.4a: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16K. Identical initial population is basis for comparison.	389-393
C.4b: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16K. Evolutionary progress is basis for comparison.	394-398
C.4c: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 16K. Location of optimal individual is basis for comparison.	399-403
C.5a: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 64K. Identical initial population is basis for comparison.	404-408
C.5b: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 64K. Evolutionary progress is basis for comparison.	409-413

Figures continued

	page
C.5c: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 64K. Location of optimal individual is basis for comparison.	414-418
C.6a: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16K. Identical initial population is basis for comparison.	419-423
C.6b: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16K. Evolutionary progress is basis for comparison.	424-428
C.6c: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 16K. Location of optimal individual is basis for comparison.	429-433

Chapter 1

Introduction and background

Genetic Algorithms (GAs) are modelled after Darwinian evolution. They are used for two purposes: 1) to study the evolutionary process, and 2) as a search procedure. In the former case, scientists focus on the dynamics of the process and the properties of the evolving population(s). In the latter case, researchers simulate, utilize, and modify a process which has been operating in the natural world.

The typical operand for a genetic algorithm (GA) is a bitstring. The bitstring, void of meaning, is called the *genotype* or *genome*. The *phenotype* is the functionality or expression of the bitstring. When researchers decide to use a genetic algorithm to find a solution to a problem, they must first determine the information which is needed in the phenotype. Then the phenotype-genotype mapping must be designed. Each functional piece of information, referred to as a *field* in the genome, is represented as a bitstring. The bits of each field are then assigned fixed locations along the genome. The mapping or placement of the bits of each field into the genome can greatly affect the convergence of the genetic algorithm.

1.1 Examples of phenotypes

This section provides examples of different types of phenotypes which have

been used. Those used in evolutionary computation for artificial life and artificial intelligence experiments fall into several categories. Among them are problem-specific genomes, genomes representing neural networks, *Tierran* genomes coded in a very simple assembly language (Ray 1992), Koza's (1992) genetic programming paradigm (GPP) in which genomes represent LISP S-expressions, and genomes representing finite state machines (FSMs).

One example of a problem-specific phenotype was used by Hillis (1992) who employed evolutionary computation to breed a minimal sorting algorithm. The phenotype consisted of paired elements in the order in which they were to be compared; this phenotype is not applicable to other types of problems. Similarly, Dawkins' (1986) *Blind Watchmaker* breeds *biomorphs*; his genotype is translated into two-dimensional pictures. The *biomorphs* are the resulting phenotype—the interpretation, functionality, or physical expression of the bitstring/genotype. In "The Evolution of Evolvability," Dawkins (1989) describes how he developed the genotype. The phenotype, the graphical interpretation of the bitstring, yielded a portfolio of pictures, some of which bear some resemblance to living organisms (Dawkins 1989, Levy 1992). But it is clear from Dawkins' description that his genome is unique to his biomorphs.

Because neural networks model solutions to many types of problems, they are frequently used as the phenotype for an evolutionary process. When the node connections of the net are predetermined, only the weights are needed in the genome (Jefferson et al. 1992, Paredis 1994); but sometimes the structure of the net is also developed by the evolutionary process (Belew et al. 1992, Collins and Jefferson

1992). These two different approaches require different ways to represent a neural net in a genome.

In Ray's *Tierra*, the genome is interpreted as a program in the *Tierran* language. (See Maley (1994) for a complete description of the *Tierran* language.) While the original system was developed as an evolutionary experiment to observe the types of self-replicating programs which would evolve (Ray 1992), more recently Ray (1994, 1998) proposed that *Tierra* be installed on the Internet to see what types of useful self-replicating application programs might evolve. With neural nets and the *Tierran* language, the phenotype is not problem-specific.

Similarly, Koza's genetic programming paradigm (GPP) provides a universal phenotype. In the GPP a representation of LISP S-expressions form the genome. Koza (1992) treats the S-expressions as trees and defines the crossover and mutation operators in order to maintain a population of valid S-expressions. The GPP has been used to solve a wide variety of problems, including locating an artificial ant which can follow an evaporating pheromone (chemical) trail (Koza 1992), and generating a pseudo-random number generator (Koza 1991). Reynolds bred tag players (Reynolds 1994b) and vision-based obstacle avoidance behavior (Reynolds 1994a), and Angeline and Pollack (1994) bred winning tic-tac-toe strategies using the GPP.

Another way to represent the solution to a problem is with a finite state machine (FSM). This data structure has also been used to study diverse problems. Fogel (1991), Angeline (1994), and Stanley et al. (1994) used FSMs to explore the iterated prisoner's dilemma. MacLennan (1992) represented his *simorgs* (simulated

organisms) as FSMs to study communication development. (MacLennan's work is interesting in that he incorporates learning into the model. Generations overlap and learning is passed on from generation to generation.) And Jefferson et al. (1992), and Angeline and Pollack (1993) used a finite state machine genome to breed an artificial ant capable of following an evaporating pheromone trail (as Koza 1992 above). As the successful FSM ant found by the GA of Jefferson et al. (1992) traverses the trail, it becomes obvious that evolutionary computation might find an unexpected solution to a problem, rather than a logical one. (Section 4.2 presents the trail problem and some general trail following strategies. Chapter 6 contains examples bred by the GA.)

"The evolutionary process can be applied to problems where heuristic solutions are not available or generally lead to unsatisfactory results" (Fogel 1994a, p. 38), due to intractability. Evolutionary computation can overcome the limitations of a person looking for the solution to a problem (Fogel 1994b), since it explores the set of possibilities without human intervention and bias. It is, however, limited by the choice of both the fitness function and the genome. A poor choice for a fitness function can drive the process to an inappropriate solution to the problem at hand (Thearling 1994). Similarly, if a problem requires at least a 10-state FSM for a solution, but the genome is limited to eight states, a solution cannot be found.

The basic operators used in genetic algorithms are fitness-based selection, reproduction, crossover, and mutation. However, human ingenuity and the use of phenotypes with different structures (e.g. trees, FSMs) have resulted in the development of new operators. For example, in his genetic programming paradigm

Koza's phenotype is an S-expression. Treating these as linear bitstrings and arbitrarily exchanging the tails of such expressions, as would be the case with crossover permitted anywhere along the bitstring, would frequently result in an invalid S-expression: that is, one with unmatched parentheses. Similarly, inverting a bit might create an invalid operator or undefined operand within the S-expression. Consequently, Koza redefined crossover and mutation so as to maintain the integrity of his data structure. He treats the S-expressions as trees. Crossover exchanges subtrees between the parents. Mutation replaces a subtree with a new, randomly generated one (Koza 1992).

Angeline and Pollack (1993) define a "compression" operator which freezes part of a genome and prevents that part from being altered by mutation. They also introduce an "expand" operator which "releases a portion of the compressed components so [it] can once again be" (p. 4) subject to alteration. These operators are defined for, and applied to, FSMs, and S-expressions within the GPP. The compression operator arbitrarily selects a subroutine. When such a subroutine is useful it tends to be retained and increasingly represented in the population, while less useful ones die out. The authors approach the artificial ant problem with an FSM genome and report faster results: that is fewer FSMs were evaluated and fewer generations were needed to find a successful ant with compression and expansion than without it.

The work by Jefferson et al. (1992), and Angeline and Pollack (1993) raises questions about the efficiency of using a GA to breed finite state machines. Since there are algorithms (operators) which can be used to reduce an FSM or change its

appearance, and since genetic algorithms are being used to search an FSM space to solve different types of problems, finding ways of making the process more *efficient* for an FSM genome would be useful. The different methods, which are used to evaluate efficiency, are described in chapter 7.

In the next section, the FSM as the phenotype is discussed in more detail.

1.2 Finite state machines as the phenotype

The initial impetus for the research described in this thesis was the work done by Jefferson et al. (1992) with their FSM ants. They defined an artificial ant as an FSM and bred them to be able to follow a given trail representing a dissipating pheromone trail. Ants lay down a chemical (pheromone) trail when returning from a food source in order to direct other ants in the colony to the foodstore. Since the trail is put down as an ant returns from the food, the part of the trail farthest from the home base has partially dissipated, and tends to be weakest and more likely to contain unmarked sections. The trail used by Jefferson et al. (1992) mimicked this effect. Their genetic algorithm did find FSM ants that could complete the trail in the time frame imposed, but it left open the question of what modifications might enhance the search process.

The genome used by Jefferson et al. (1992) had 453 bits (described in section 4.2) and therefore defined a space of 2^{453} bitstrings. This space can be divided into equivalence classes with each class containing equivalent FSMs. For example, the two machines in figure 1.1 are equivalent: q_1 and q_2 have simply exchanged names. Since there are many equivalent FSMs, a number of variations of a nearly successful

FSM may exist in a population, but the similarity may not be evident due to different representations. The GA used by Jefferson et al. (1992) does not utilize this fact to build and retain useful segments of the genome (schemata) called building blocks. Based on schema theory (section 2.2.4), useful schemata in the FSM genome compete with each other during the selection process, each hampering the growth of the other.

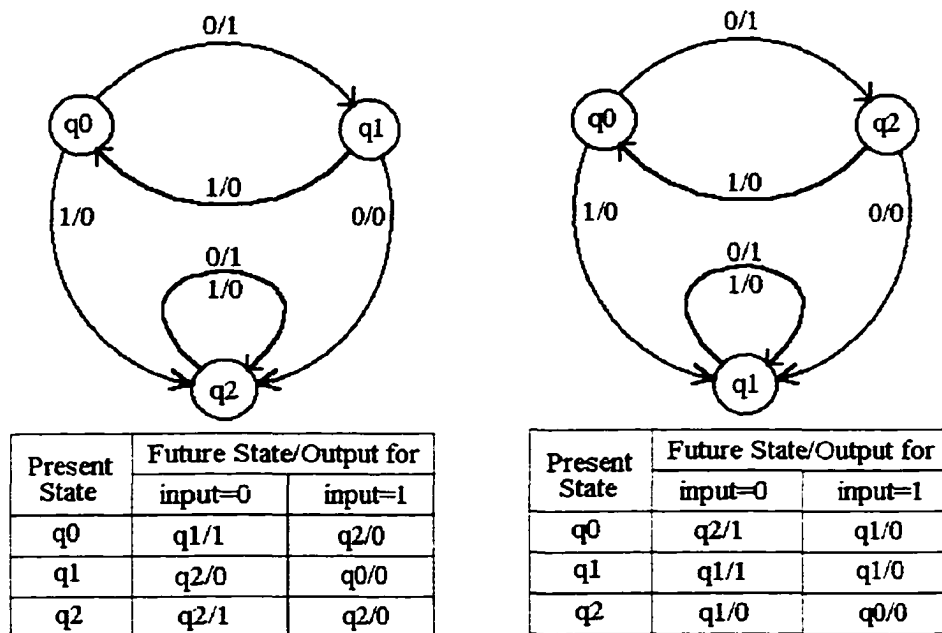


Figure 1.1: Equivalent FSMs

Jefferson et al. (1992) allowed for a 32-state FSM. Any one of the 32 states could be the start state. Similarly, the other states of an FSM could be any of the remaining states. Therefore the 3-state FSM of figure 1.1 has $32 \times 31 \times 30 = 29,760$ different representations in the Jefferson et al. (1992) genome. The nature of the

genome used by Jefferson et al. (1992) allows for a extensive exploration of the genome space when the genetic algorithm is applied, but the nature of both the genome and the FSM also seem to hamper the growth of useful schemata.

While their research was fruitful, as indicated earlier, GA schema theory and the finite state machine genome implemented by Jefferson et al. (1992) appear to be incompatible. In theory, the genetic algorithm moves a population towards improved solutions (Goldberg 1989) as useful schemata (useful segments of a genome) grow in number and length. Because a single finite state machine has different representations (simply by changing state names), identical machines with different representations do not necessarily share a useful schema: that is a set of identical transitions/outputs which provide a successful strategy could reside in different parts of the genome, and appear very different due to having different state names. Consequently, these machines hamper schema growth as the machines compete against each other for their share of the next generation. The research for this thesis studies the effect of two new operators/algorithms, called SFS (Standardize Future State) and MTF (Move to Front), which reorganize the machines during run time so identical machines with the same number of states will appear the same (chapter 5). The purpose of these new operators is to enhance the growth of useful schemata and thereby hasten the convergence of the genetic algorithm. The strategies these operators employ are FSM-specific, and theoretically should improve the convergence of the genetic algorithm for FSM genomes. Experiments were performed with these new operators in order to determine how they affect the efficiency of the genetic algorithm.

Chapter 2 provides relevant definitions, introduces the genetic algorithm, and discusses its relationship to evolutionary computation and the evolutionary algorithm. Chapter 3 presents the research project, and chapter 4 introduces the benchmark case to which the reorganizing procedures are added, and to which results are compared. The reorganization algorithms are presented in chapter 5, and chapter 6 reviews some interesting machines which were bred by the genetic algorithm during experimentation. The experimentation procedures are introduced, and the theory and the strategies which will be used to evaluate the effect of the new operators on the efficiency of the genetic algorithm are developed in chapter 7. In chapter 8, the experimentation procedures are presented in more detail, and the evaluation of the performance of the reorganization operators appears. Chapter 9 analyzes the performance of the new operators across a range of problems, and the conclusions and directions for future work appear in chapter 10.

>>>●<<<

Chapter 2

The genetic algorithm

This chapter introduces a simple genetic algorithm, reviews some variations of it which have appeared in the literature, and presents relevant theory. The operand for a genetic algorithm (GA) is typically a bitstring. The bitstring, void of meaning, is called the *genotype* or *genome*. The *phenotype* is the functionality or expression of the bitstring. When using a GA to solve a problem, the information required for the phenotype must first be determined. Then the phenotype-genotype mapping must be designed. Each functional piece of information, referred to as a *field* in the genome, is represented as a bitstring. The bits of each field are then assigned fixed locations along the genome. The mapping or placement of the bits of each field within the genome can greatly affect the convergence of the genetic algorithm. A *fitness function* must then be defined to evaluate the ability of each phenotype to survive in a given environment. Often this quantifies the ability of the phenotype to perform some task, for example to follow a trail (Jefferson et al. 1992, Koza 1992, Angeline and Pollack 1993).

2.1 A simple genetic algorithm

The genetic algorithm treats bitstrings (or other data structures) as genomes

(chromosomes). A GA is generally started with a population of randomly generated bitstrings (genomes, individuals). Each genotype (bitstring) has an associated phenotype, which is the interpretation or functionality of the bitstring. Using the phenotype, each individual is assigned a fitness value using some predefined function or procedure. Operators, generally biologically-based, such as fitness-based *selection*, *reproduction* (cloning), *crossover* (mating), and *mutation* are applied to a population of bitstrings in order to breed a new population. *Selection* determines which bitstrings will be copied into and used as the parents of the next generation. *Reproduction* copies (clones) the selected individuals to parent the next generation. *Crossover* is a mating operation in which genetic material (parts of two bitstrings) is exchanged between two parents to form the genome of the offspring. Finally, *mutation* arbitrarily alters the genotype, generally by randomly changing the value of one or more bits. After mutation, the new generation is in its final form (Goldberg 1989).

The following is an outline of a typical genetic algorithm.

- 1) Create the initial generation as the present generation.
- 2) Find the fitness of each member of the present generation using a predefined function or procedure.
- 3) Form the initial phase of the next generation by selecting mating pairs and copying them into the next generation.
- 4) Generate the intermediate phase of the next generation from the initial phase by exchanging genetic material (performing a crossover operation) between the individuals of each mating pair.

5) Apply mutation to the intermediate population to get the final phase of the next generation. At this point this becomes the present generation. The research described in chapters 3 and 5 reorganizes these genomes before the newly formed population parents the next generation.

6) Repeat from step 2 until the criteria for termination is satisfied.

The operations in steps 3, 4, and 5 are not deterministic; they are implemented with respect to some probability density function. For example, individuals are often selected for reproduction with a probability proportional to their fitness: the more-fit individuals have a correspondingly higher probability of being selected. Mutation of the genomes occurs with some fixed probability. For example if the mutation rate is 0.5% per bit, and the population contains 100 individuals with 50 bits per individual, one would expect $.005 \times 50 \times 100 = 25$ bits to be changed. Because the genetic algorithm is implemented as a stochastic (probabilistic) model, there may be more or less than 25 bits altered. The following very simple example will clarify this.

Consider a problem with the genotype consisting of six bits. The phenotype views the bitstring as a binary integer, and the fitness is the value of this integer plus one. For example, the genotype 101100 has a phenotype of 101100_2 , that is the same six bits are viewed as a binary integer. The decimal equivalent of 101100_2 is 44, therefore its fitness is $44+1=45$. The population contains six individuals, and mutation is applied with a probability of 0.1 per bit. This problem has been chosen to illustrate the operation of a genetic algorithm because its solution is obvious, and because it is easy to observe the GA's progress towards the solution.

To start the genetic algorithm for this problem, randomly generate 36 bits to form the initial population of six individuals consisting of six bits each. A run generated the results shown in table 2.1. The maximum fitness for this population is 53 and the average fitness is 35.67.

Generation 0

Individual #	0	1	2	3	4	5
Genotype (bitstring)	101100	000011	110100	100101	100010	100110
Phenotype (binary integer)	$101100_2 = 44_{10}$	$000011_2 = 3_{10}$	$110100_2 = 52_{10}$	$100101_2 = 37_{10}$	$100010_2 = 34_{10}$	$100110_2 = 38_{10}$
Fitness	$44 + 1 = 45$	$3 + 1 = 4$	$52 + 1 = 53$	$37 + 1 = 38$	$34 + 1 = 35$	$38 + 1 = 39$
Fitness: partial sums	45	$45 + 4 = 49$	$49 + 53 = 102$	$102 + 38 = 140$	$140 + 35 = 175$	$175 + 39 = 214$
Fitness range for selection	1–45	46–49	50–102	103–140	141–175	176–214

Table 2.1: Initial Population

The creation of the next generation occurs in three steps:

- Using fitness-based selection choose and copy the parents into the initial phase of the next generation.
- Form an intermediate population by pairing the parents of the initial phase into mating pairs. The crossover operator exchanges genetic material (bits) between the parents in each mating pair, producing two offspring from each mating pair.
- Apply mutation to the intermediate population to get the final population for the next generation.

To get a fitness-based selection for generation 1 as indicated in step a above, determine the partial sums of the fitnesses as indicated in table 2.1. This is used to define a fitness range for each individual: that is a unique range of integers is associated with each individual. The individual fitness ranges are contiguous. This defines a total fitness range for the selection process with each integer in the range corresponding to an individual in the population and with all individuals in the population included in the range. The individual fitness ranges reflect the relative fitness of the individuals with more-fit individuals having a correspondingly larger part of the total fitness range.

As indicated in table 2.1 the complete fitness range for generation 0 is 1 to 214. Since six individuals are needed to parent the next generation, generate six random integers between 1 and 214 in order to implement the fitness-based selection. Each of the six numbers is used to select an individual to be duplicated (reproduced) into the first phase of the new generation. For this example an integer between 1 and 45 selects individual #0, between 46 and 49 chooses individual #1, and similarly across the last row of table 2.1. The result of this selection process appears in table 2.2. It is interesting to note that in generating the initial phase of generation 1 the least-fit individual from generation 0 (#1 from table 2.1) was not selected, and one of the more-fit individuals (#0 from table 2.1) was selected twice. As a result the most significant bit of all individuals is now one. This is quite advantageous as this is what the most significant bit will have to be to maximize the fitness; the selection process has started to move the population towards the optimal solution.

Generation 1: Initial phase

Individual # in generation 1	0	1	2	3	4	5
Random # for selection	71	168	103	25	197	37
# of individual selected from generation 0 of table 2.1	2	4	3	0	5	0
Corresponding genotype from table 2.1	110100	100010	100101	101100	100110	101100

Table 2.2: First Generation after Selection

To generate the second phase for this population, the individuals from the initial phase of generation 1 are paired for mating in the order in which they were selected: that is individuals 0 and 1, 2 and 3, and 4 and 5 from table 2.2 are paired, with each pair ultimately producing two children. Three random integers are then generated for crossover points, one number for each mating pair. These integers indicate the number of bits to retain from each individual in the corresponding pair: the remainder of the strings are exchanged. To ensure that some genetic material is exchanged, the integers used for crossover must result in the retention of at least the first bit of the parents, and the exchange of at least the last bit of the parents. A one will retain the first bit, and a number which is one less than the length of the genome will exchange the last bit. Since the genome in this example consists of six bits, the integers generated must be between one and five inclusive. The numbers generated for the mating process are 3, 3, and 5. The crossover (mating, exchange of genetic material) is carried out in table 2.3. The space in each pair of bitstrings

indicates the crossover point. The bits after the crossover point are exchanged. Note that numbers 4 and 5 in table 2.3 are not altered by the crossover.

Generation 1: Intermediate phase

Pair	Individual #	Before Crossover	After Crossover
1	#0	110 100	110 010
	#1	100 010	100 100
2	#2	100 101	100 100
	#3	101 100	101 101
3	#4	10011 0	10011 0
	#5	10110 0	10110 0

Table 2.3: First Generation after Crossover

To apply mutation to this intermediate population generate 36 random numbers between 0 and 1 with each number corresponding respectively to each of the 36 bits in this intermediate population. To implement the mutation rate of 10%, change the value of a bit if the corresponding random number is less than 0.1. With 36 bits and a 10% mutation rate, it is expected that (0.1×36) three or four bits would be mutated. Applying mutation produced the results indicated in table 2.4. Note that four bits are mutated—those in boldface and underlined. The line labelled "After mutation" is the final population for generation 1. It has a maximum fitness of 61 and an average fitness of 45.83. In one generation (tables 2.1 and 2.4), this simple genetic algorithm has come close to finding the optimal individual and the population as a whole has shown a significant increase in fitness.

Generation 1: Final phase

Individual #	0	1	2	3	4	5
Before mutation	110010	100100	100100	101101	100110	101100
After mutation	110 <u>1</u> 10	100100	100100	1 <u>1</u> 110 <u>0</u>	100110	10110 <u>1</u>
Fitness after mutation	55	37	37	61	39	46

Table 2.4: First Generation after Mutation

While in a single generation the results need not be so impressive (See figure 2.1, generations 2 to 3.), over many generations the trend is towards increasing maximum and average fitness. Figure 2.1 summarizes the results of a sample short run for this problem. Note that while both the maximum fitness and average fitness do decrease at times, the general trend is increasing. This genetic algorithm was executed for 10 generations. It found and retained the optimal individual, but this is not necessarily the case. A genetic algorithm will not necessarily locate an individual with the optimal fitness.

A GA generally will find an individual with a fitness which is close to the optimal value and acceptable for the problem at hand. For example, a GA may be used to determine the structure and initial weights of a neural network; then a learning algorithm can be used to finalize the weights (Belew et al. 1992). There are problems (such as the travelling salesman problem) for which the optimal solution cannot be found without an extensive search using an algorithm of unacceptable complexity and for which the fitness value of the optimal individual in a genetic algorithm is not known (Nilsson 1980, Goldberg 1989).

Generation	Phenotype for individual #						Maximum	Average
	0	1	2	3	4	5	fitness	fitness
0	39	50	41	50	19	23	51	38.00
1	50	18	19	47	51	42	52	38.83
2	47	19	43	62	55	41	63	45.50
3	23	1	15	31	39	55	56	28.33
4	55	55	55	38	7	55	56	45.17
5	23	55	47	54	50	39	56	45.67
6	51	15	51	46	34	59	60	43.67
7	34	33	59	59	59	59	60	51.50
8	59	3	59	43	58	59	60	47.83
9	63	59	59	59	8	56	64	51.67
10	63	59	59	8	63	56	64	52.33

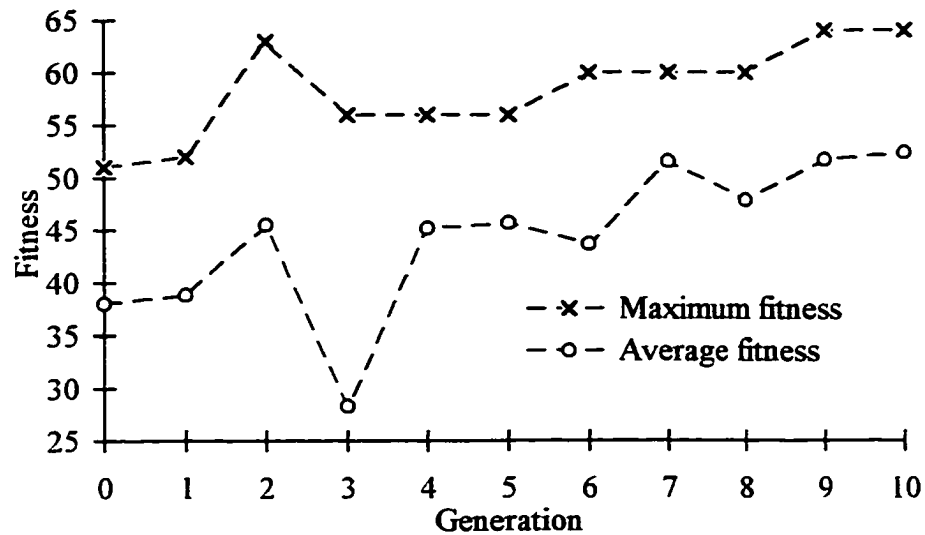


Figure 2.1: Sample Run

A genetic algorithm is used for applications in which a near optimal solution is acceptable. For example, Hillis (1992) used a genetic algorithm to try to find a minimal sorting-network to sort a list of 16 items. After several attempts and modifications (described in section 2.2) of his genetic algorithm, a 15-minute run on his connection machine resulted in a 62-comparison network. A subsequent run

reduced the number of comparisons by one. Levy (1992) documents that in 1969 a 60-comparison network was published. Using evolutionary computation, Hillis (1992) was able to find an algorithm which performed almost as well as the best known procedure (Hillis 1992, Levy 1992).

It is obvious that Hillis' modified GA found a suboptimal solution because the 60-comparison network was published, but in turn the 60-comparison network could also be suboptimal. Perhaps new techniques can be injected into Hillis' last GA. techniques which might force the GA to explore the genome space in a different way and which can locate a better solution.

Variations on the operators of selection, crossover, and mutation as they are implemented in this section, are documented in the literature. Some of these variations are presented in the next section.

2.2 Variations on the simple genetic algorithm

There are many variations of the simple genetic algorithm. Hillis' (1992) GA. discussed in the previous section, incorporated one possible variation. One of Hillis' modifications was the installation of a second evolving population or species which challenged his sorting-networks. This produced a changing environment for the sorting-networks to master. The fitness for an individual was not a fixed value, but was determined by how well an individual fared against a changing competitor. This exemplifies an alternate way to define fitness. It is significantly different from the explicitly defined fitness function used in the example developed in the beginning of the previous section. In that example, the fitness for an individual was constant and

the population was moving towards a fixed optimal fitness value. In addition, the example developed in the beginning of the previous section demonstrates one way to implement the biologically-based operators of selection, crossover, and mutation. Other operators, and numerous variations of the biologically-based operators have been used. The next section presents different ways to define fitness. In the sections which follow, variations of the basic operators are introduced in addition to other operators, some of which are not biologically based.

2.2.1 Variations on fitness

In the example presented in the beginning of the previous section, the fitness function was defined explicitly yielding a single numerical value for the fitness, a value which is a constant for an individual. In another example of explicitly defined fitness, the bitstring is interpreted as an algorithm and the fitness function quantifies an individual's ability to perform some task, for example to traverse a trail (Jefferson et al. 1992, Koza 1992).

Alternately, the fitness can define an individual's relationship to a changing environment. In this case the individual's fitness varies with the environment. One way to implement this is to hold a competition between the individuals in a population. For example, Reynolds (1994b) interprets a bitstring as a strategy to catch and evade an opponent when playing the game of tag. Individuals compete against each other and the fitness measures the percent of the time that an individual avoids being the pursuer during the game. Several researchers, for example Axelrod (1987) and Fogel (1991), experimented with the iterated prisoner's dilemma. In this

problem, the fitness is based on different scores for mutual cooperation, mutual defection, and a cooperation-defection situation when different strategies in the evolving population play against each other. In these two different applications, the fitness measures an individual's relationship to its changing environment, where the environment consists of the changing population.

Another way to create a changing environment is by breeding opposing populations. As previously mentioned, Hillis (1992) bred a population of sorting algorithms. After initial testing, he installed an opposing population (species) of permutations of numbers which were to be sorted. In this case each species created the environment for the other. Each member of each population faced a fixed number of competitions against individuals of the opposing population. The fitness of each sorting-network was the number of competing permutations which it could sort, and the fitness for each permutation was the number of sorting-networks which could not sort it. Hillis noted that each population evolved to exploit the weaknesses of, and avoid the strengths of the other. This method, which is similar to a predator-prey situation in nature, made it unnecessary to test each sorting-network on the $16!$ permutations of 16 items in order to determine the fitness of each network.

Another variation for fitness is presented in *Tierra*, in which an evolutionary algorithm (EA) is applied to self-replicating programs. (See section 2.2.7 for a comparison between GA and EA.) The individuals are placed in *Tierra's* Reaper queue; those individuals at the front of this queue, generally the older ones and those unable to self-replicate, are eliminated when memory becomes too crowded. In this system, fitness is defined by the Reaper. Ray, who is a tropical ecologist,

documented the development of an ecological system by the computer similar to those found in nature (Ray 1992).

Evolutionary computation (including genetic algorithms) is used for two different types of research: 1) to study the changing properties of an evolving population, and 2) as a search algorithm. The difference between these two forms of research is important because the definition of the fitness function is different. Ray (1992) investigated the emergent properties of a population during evolution. Jefferson et al. (1992) and Koza (1992) were looking for a solution to a problem. Hillis (1992) was interested in observing the evolutionary process and used the GA as a search procedure to do so (Levy 1992). When a researcher's main purpose is to study the changing properties of an evolving population, an environment defines the individual's fitness; changing the environment during execution changes an individual's fitness during a run. A changing environment can also be used when a GA is used as a search procedure.

While variations in defining the fitness function are somewhat dependent on the problem, variations in the selection process are not. Some alternatives in implementing the selection process are reviewed in the next section.

2.2.2 Variations on the selection process

In a genetic algorithm a newly bred population may completely or partially replace the old one when forming each subsequent generation. In the example presented in section 2.1, the fitness-based selection which was used is called the roulette wheel method. An individual could be selected multiple times, as shown in

table 2.2; in fact the more-fit individuals generally are selected more frequently (Goldberg 1989). In an alternative to the roulette wheel method, the population can be sorted by fitness and only the most-fit (for example the top 5%) are permitted to reproduce. Once this group is chosen, each individual in the group has an equal chance of being selected to parent each child (Jefferson et al. 1992). If an optimal individual is found by a genetic algorithm using either of these selection algorithms, it can be lost since the whole population is replaced by each subsequent generation. If an individual seems to appear in two adjacent generations, it is not due to the survival of the individual, but rather to the development of an identical one.

Other selection methods copy one or more of the most-fit individuals into the next generation and these clones are not subject to further operations. In these selection processes the bitstrings are ordered by fitness and the least-fit is/are replaced. In the *genitor* method, only the single, least-fit individual is replaced, one at a time (Whitley 1989a and 1989b). In the *elitist* method, only the most-fit individual is cloned into the next generation (De Jong 1975, Goldberg 1989). Between these two extremes of replacing only one or all but one individual at a time, the selection process can replace some part, for example the bottom half, of each generation. This produces *overlapping* generations, that is sequential generations which share unaltered genomes. With these partial replacement methods, an optimal individual will not be lost since the most-fit from each generation is/are carried over into the new one. With the retention of the most-fit in a population, there is often a more consistent climb of the average fitness of the population.

This section presented variations in the selection process. In the next section,

new operators, and variations on the biologically-based operators of crossover and mutation are discussed.

2.2.3 Crossover and mutation variations, and other operators

In addition to a wide variety of selection methods previously discussed, the literature also contains variations on the other two biologically-based operators, in addition to operators which are not biologically-based. The basic operators used in genetic algorithms are fitness-based selection, reproduction, crossover, and mutation. Human ingenuity and the use of phenotypes with different structures (for example trees, FSMs) have resulted in redefining the basic operators and the development of new operators. Levinson (1994) experimented with the conversion operator. Gene conversion is "defined as replacement of a random segment of random string a by a corresponding segment from random string b , without altering string b " (Levinson 1994, p. 91). For example, the middle three bits of 11100 can replace the center three bits of 00110, replacing the latter individual with 01100—but unlike with crossover the 11100 remains in the population unaltered.

Jefferson et al. (1992) bred finite state machines (FSMs) and implemented a variation of the crossover operator presented in section 2.1. Instead of selecting a single crossover point, they applied a 1% crossover rate per bit. Two parents were selected (call them P1 and P2) with one of the two (say P1) chosen as the bit donor. After each bit was copied from the genome of donating parent P1 into the child's genome, a random number was generated. When the random number was less than 0.01, parent P2 became the donor. When the random number generator again

produced a number less than 0.01 the bit donor was changed back to P1. Essentially each random number less than 0.01 changed the donating parent, thus producing a crossover point. With this crossover algorithm, zero to several crossover points can result. As implemented by Jefferson et al. (1992) only a single child was produced from the mating.

Koza (1992) defines crossover and mutation so as to maintain the integrity of his data structure. In the genetic programming paradigm (GPP), the phenotype is an S-expression, a data structure which is easily destroyed by previously defined crossover and mutation operators. To prevent breeding from yielding invalid S-expressions, in the GPP S-expressions are viewed as trees with crossover exchanging subtrees between the parents, and mutation replacing a subtree with a new, randomly generated one.

Angeline and Pollack (1993) define a "compression" operator which freezes part of a genome and prevents that part from being altered by mutation. They also introduce an "expand" operator which "releases a portion of the compressed components so they can once again be" (p. 4) subject to alteration. Angeline and Pollack created the new operators with the expectation of enhancing the performance of the evolutionary algorithm based on the underlying theoretical concepts explaining why evolutionary computation succeeds as a search procedure. Similarly, the reorganization operators, which were created for the GA research presented in this document and which are fully described in chapter 5, were designed with the same goals in mind. The underlying concepts explaining the success of the genetic algorithm as a search procedure are presented in the next section.

2.2.4 Schema: definition and theory

The success of the genetic algorithm as a search procedure is often believed to be based on the formation and proliferation of useful building blocks called *schemata*. This section covers general schema theory. The next section presents an example which demonstrates the theory.

A *schema* is a template which indicates the similarities in strings. The alphabet for schemata for bitstrings is $\{0, 1, *\}$. A bitstring matches a schema if it contains a zero in each position that the schema has a zero and a one in each position that the schema has a one. An asterisk in a schema acts as a place holder and does not identify the bit required in the given position of a matching bitstring: that is a bitstring will match a position regardless of whether it contains a one or zero in the schema position containing an asterisk. For example, 10011 and 11010 have a maximum match represented by the schema $1*01*$. This schema matches four bitstrings—10010 and 11011 in addition to the two already given. Replacing any zero or one in this schema with an asterisk will result in a schema which still matches all four individuals, but since each of the asterisks can match a zero or one in a bitstring, the schema with three asterisks will match a total of $2 \times 2 \times 2 = 8$ bitstrings.

The *fixed* positions of a schema are those positions containing a zero or one. The *defining length* of a schema is the difference between the first and last positions containing a symbol other than an asterisk, or alternatively the number of points between the first and last fixed bits of the schema at which crossover can occur. As suggested by Goldberg (1989), it is the building of useful schemata with increasing length within a population that moves the population towards a solution. Looking

at figure 2.1, in generation 0, two individuals match 11^{****} and none match 111^{***} . By generation 2, one individual matches 111^{***} .

The defining length of a schema affects the chance of its disruption due to crossover. Let crossover exchange the tails (consisting of a randomly chosen number of bits) of two bitstrings. If the schema ab^{***} ($a, b \in \{0, 1\}$) matches individuals with high fitness, there is a one in four chance that a crossover point will come between these two fixed bits of an individual bitstring which matches it. On the other hand if $a^{***}b$ ($a, b \in \{0, 1\}$) matches highly fit individuals, any crossover on an individual matching this schema will separate these two fixed bits. When two individuals matching a schema are crossed, the two children will match the schema. For example, 111010 and 110111 match 11^{**1*} ; any crossover point on these two bitstrings will create two individuals which still match this schema.

Useful schemata are those which match highly fit individuals. It is the growth of both the number and defining length of useful schemata within a population which drives the population as a whole toward higher fitness. Looking at figure 2.1, four individuals in generation 0 match 1^{*****} , and in generation 2, five individuals match it. 11^{*****} is another useful schema for this problem. In generation 0 two individuals match it. Only one bitstring matches it in generation 3 when both the maximum and average fitness decrease. This increases to four matches in generation 4 when the maximum fitness retains its value and the average fitness surges back almost to its prior level. Comparing the populations in tables 2.1 and 2.4, five individuals in generation 0 match 1^{*****} ; by generation 1, all six match it. As was pointed out earlier, this is beneficial as the final solution must match 1^{*****} .

The theory offered by Goldberg (1989) explaining the success of the genetic algorithm as a search procedure focusses on schemata growth—not on the individual bitstrings which are the operands of the GA. A genetic algorithm, while breeding individuals, processes many more schemata. For example for a seven-bit genome the genome space consists of 2^7 individuals which match 3^7 schemata. A single seven-bit genome matches 2^7 schemata as each position in a schema can contain either the bit which fills the position in the genome or an asterisk. The theory underlying the progression of a population towards a solution to a problem is based on determining the expected growth in the number of useful schemata during the execution of a genetic algorithm.

In general, let f_s be the fitness of an individual which matches schema S . $E(S, g+1)$ be the expected number of matches to schema S in generation $g+1$. $N(S, g)$ represent the number of matches to schema S in generation g , and f_i be the fitness of the i^{th} individual in generation g . When using the dart board method of selection, the probability that a selected individual matches schema S is $\frac{\sum f_{s_j}}{\sum f_i}$, where $\sum f_{s_j}$ is the sum of the fitnesses of those individuals in the population which match S and $\sum f_i$ is the sum of the fitnesses over the entire population. Since a match to schema S can occur with this probability when each individual is selected to fill the n members of the next generation,

$$E(S, g+1) = n \frac{\sum f_{s_j}}{\sum f_i} = \frac{N(S, g) \frac{\sum f_{s_j}}{N(S, g)}}{\frac{\sum f_i}{n}} = N(S, g) \frac{f_{s_{AV}}}{f_{AV}}. \quad (\text{Equation 2.1})$$

In the above equation $f_{s_{AV}}$ is the average fitness of members of generation g which

match S and f_{AV} is the average fitness of generation g . From equation 2.1 it is obvious that if schema S maintains an above average fitness, the number of representatives it has in the population can be expected to increase with each successive generation. But this equation does not take into consideration the loss of matches to schema S due to crossover and mutation (Goldberg 1989).

To determine the effect of crossover and mutation on $E(S, g+1)$, let d be the number of crossover points between the first and last fixed positions in schema S . Let c be the number of crossover points in the genome; this is equal to one less than the length of the genome. If both parents match schema S , S will not be lost during crossover. Therefore the probability that a match to schema S is lost due to crossover is $\leq \frac{d}{c}$. Hence the probability that a schema survives crossover is $\geq 1 - \frac{d}{c}$. Let p_m be the probability of mutation of each bit. p_m is generally very small; usually p_m is much less than 1; frequently it is less than or equal to 0.01. Then the probability that a bit is not mutated is $1 - p_m$. Let b represent the number of fixed positions in S . The probability that S survives mutation equals the probability that none of the b fixed bits is mutated $= (1 - p_m)^b \geq 1 - bp_m$. Therefore the probability that a match to schema S survives both crossover and mutation $\geq \left(1 - \frac{d}{c}\right)(1 - bp_m) \geq 1 - \frac{d}{c} - bp_m$. This probability is largest for small d (defining length) and small b (number of fixed positions in S). In addition a new match to S might result from crossover or mutation when there are members in a population which do not match S . Keeping in mind that new matches to schema S can be born, and incorporating the probability of schema loss into equation 2.1 yields

$$E(S, g+1) \geq N(S, g) \frac{f_{SAV}}{f_{AV}} \left(1 - \frac{d}{c} - bp_m\right). \quad (\text{Equation 2.2})$$

Therefore the number of matches to S is expected to increase when S has above average fitness, short defining length, and a small number of fixed positions (Goldberg 1989). Note that when every member of a population matches S , there will be no losses due to crossover.

Based on this theory, it is the schemata, rather than the individuals in the population, which are responsible for moving the population toward a solution. The reorganization algorithms which are described in chapter 5 were designed to enhance schemata growth. One of these was designed to reduce the defining length of useful schemata. Because a short defining length is desirable, a GA using a genome in which the most significant bits are in close proximity is expected to do better than one in which the most significant bits are not adjacent to each other. In the next section an example is presented to demonstrate this point.

2.2.5 Schemata at work: an example

A simple example demonstrates the effect of the organization of the genome on a genetic algorithm in light of schema theory. Moving each run towards a solution is the growth of the useful schemata in each successive population. In this experiment the phenotype consists of two non-negative integers, x and y . x contains three bits; y has four. Therefore the genotype needs seven bits. The fitness function is $f(x,y)=x^2-y+17$. Clearly the maximum value that f can attain is 66 when $x=111_2=7_{10}$ and $y=0000_2=0_{10}$. One way this differs from the example developed in the beginning of section 2.1 is that the optimum individual contains a mixture of zeroes and ones. As in that previous example, the population contains six individuals

and the mutation rate is 10%. For this example, the maximum and average fitnesses for each generation is averaged over ten runs.

This experiment was run with two different genotype-phenotype mappings. Let $x = x_2x_1x_0$ and $y = y_3y_2y_1y_0$ where $x_i, y_i \in \{0,1\}$. In Case I the seven bits of the genotype represent $y_3x_2y_2x_1y_1x_0y_0$. In Case II they represent $y_3y_2y_1y_0x_0x_1x_2$. In the first case the most significant bits are close to each other; in the latter case, they are as far apart as the genome permits. Because a short defining length is desirable, a GA using a genome in which the most significant bits are in close proximity should do better than one in which the most significant bits are not adjacent to each other. For Case I, $\frac{d}{c}$ in equation 2.2 equals $\frac{1}{6}$ for schema 01^{*****} , that is when the most significant bits of x and y contain the bits needed for the solution. For Case II, the schema for the same condition is $0^{*****}1$, and $\frac{d}{c} = 1$ indicating that these two important bits are certain to be separated by the crossover operation.

Looking at figure 2.2, the graph on the left compares the maximum fitness reached in each generation for the two cases. The one on the right contains the average fitness in each generation for the same runs. Note that even though the first case starts with a lower maximum and average fitness than the second case, its performance surpasses that of the second case. Variations will occur, most notably in the maximum, but the average is used to determine the progress of the system and the difference there is clearly noticeable. The proximity of the significant bits in Case I aids the growth of useful schemata, moving the population as a whole towards the solution more quickly. When the bits which were more important to the solution were placed together in the genome, useful schemata had a shorter defining length,

were less likely to be lost due to crossover, and were therefore able to overtake the population faster as indicated by the big difference in the average fitness.

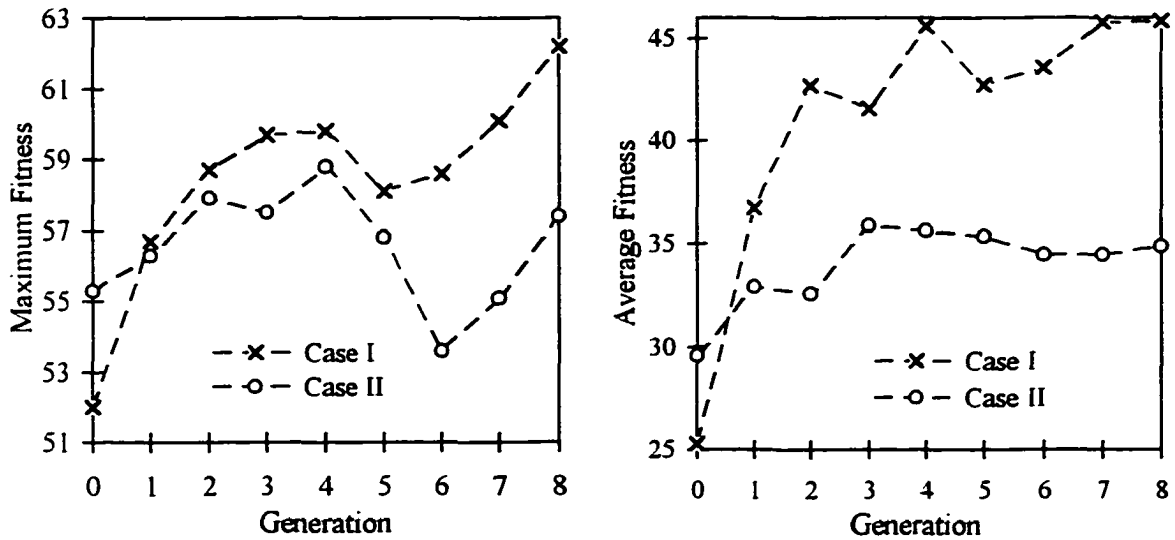


Figure 2.2: Comparison of runs with different phenotypes. Case I uses phenotype $y_3x_2y_2x_1y_1x_0y_0$. Case II uses phenotype $y_3y_2y_1y_0x_0x_1x_2$. Fitnesses are averaged over 10 runs.

But this is considered a simple problem for a genetic algorithm. If the fitness of each individual in the genome space for this problem were graphed outward from a seven-dimensional hypercube, the optimum fitness would be at the peak of a single hill. The GA used as a search procedure tries to locate this single peak. The significance of this is pursued in the next section.

2.2.6 Fitness landscape

The fitness landscape for a bitstring genome space shows the fitness of that space graphically. This is useful in determining the difficulty of a problem for a GA.

The landscape is formed by plotting the fitness of each n -bit individual in the space outward or upward from an n -dimensional hypercube, cube, or square. Each corner of these n -dimensional figures is adjacent to n other corners. The corners are labelled so that the labels for adjacent corners differ by one bit. While these hypercubes are hard to visualize for $n > 3$, the fitness landscape for a four-bit genome is easy to graph based on a two-dimensional rendition of a four-dimensional hypercube. The two-, three-, and four-dimensional figures are shown in figure 2.3.

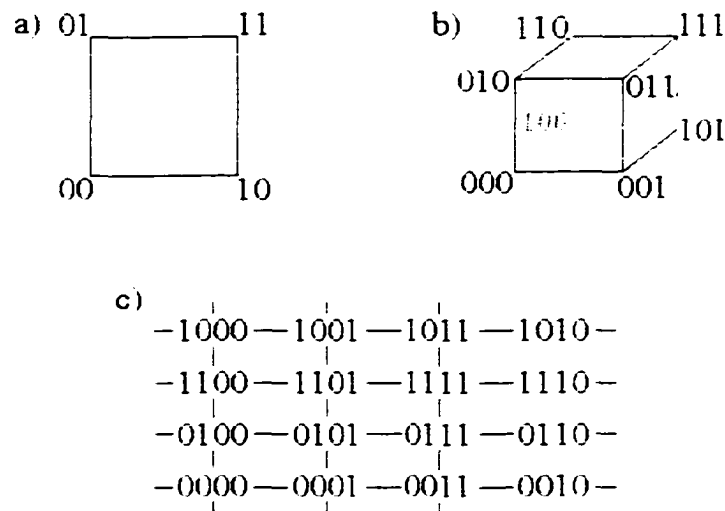


Figure 2.3: a) Two-dimensional square b) Three-dimensional cube c) Four-dimensional hypercube with top-to-bottom and side-to-side adjacency.

Fitness landscape examples appear in figures 2.4 and 2.5. For figure 2.4 refer back to the example developed in section 2.1. Reduce the genotype to four bits. Retain the phenotype as the binary integer and the fitness as one more than its decimal equivalent. Figure 2.4 contains the graphical representation of the fitness landscape for this smaller problem. Similarly for figure 2.5, reduce the problem

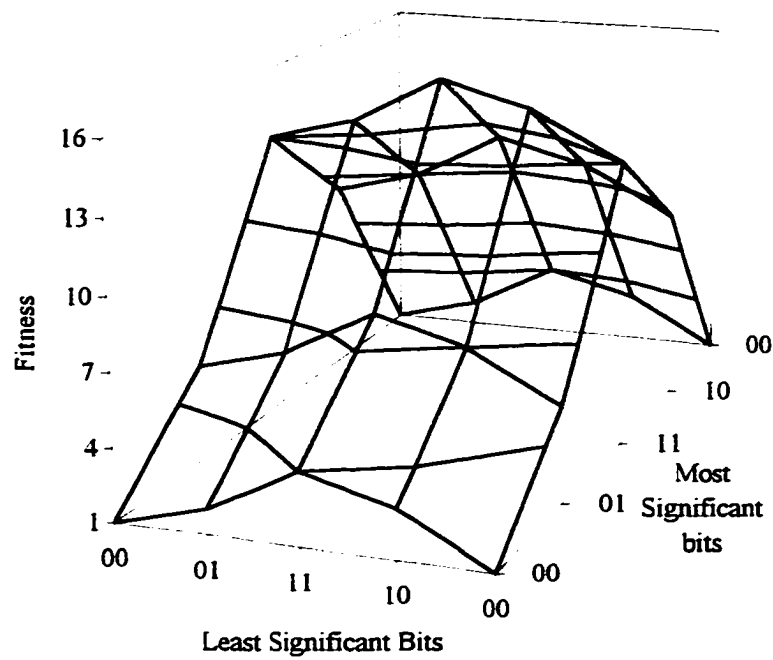


Figure 2.4: Fitness Landscape for a four-bit genome with phenotype x =a binary integer and fitness $f(x)=x+1$.

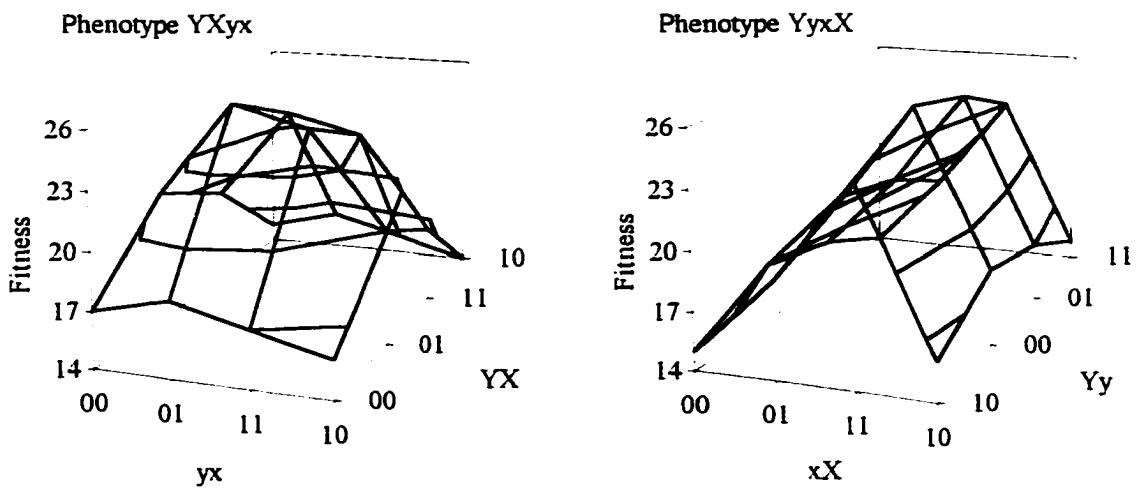


Figure 2.5: Fitness Landscape for a four-bit genotype with two different phenotypes. $x=Xx$, $y=Yy$ where $x,y,X,Y \in \{0, 1\}$, and $f(x, y) = x^2 - y + 17$

developed in section 2.2.5 to a four-bit genome, let $x = Xx$ and $y = Yy$, ($x, y, X, Y \in \{0, 1\}$), and retain the fitness function $f(x, y) = x^2 - y + 17$. Figure 2.5 contains the fitness landscapes for the smaller version of the two different phenotypes utilized in the example in section 2.2.5.

Utilizing this imagery, the most GA-friendly landscape consists of a single peak or plateau with hills sloping down to a single valley. The examples developed in sections 2.1 and 2.2.5 are instances of this as figures 2.4 and 2.5 indicate. At the other extreme are the GA-hard landscapes. These are rough and rugged, and have many peaks at different heights. Whereas in a GA-friendly landscape the evolving population of a genetic algorithm can climb the hill to the single peak, in the GA-hard case the population must be able to hop from one hill to another to avoid being stuck on a local maximum which is not an absolute maximum. The undesirable situation of being stuck on a suboptimal peak is an instance of *premature convergence* (Goldberg 1989). Premature convergence occurs when the genetic makeup of a population is homogeneous. As a result of this lack of diversity, the evolutionary process is unable to explore new areas of the genome space. Several strategies have been developed to reduce the chance of this occurring. One such strategy is used in the evolutionary algorithm (EA). In the next section the EA is described and then compared to the genetic algorithm.

2.2.7 One strategy to reduce the chance of premature convergence

There are different philosophical emphases when simulating an evolutionary process. "The term [evolutionary computation] is still relatively new and represents

an effort to bring together researchers who have been working on these and other closely related fields but following different paradigms" (Fogel 1997, p. 1). In this section some of these differences are presented in the context of a strategy to reduce the chance of premature convergence.

Genetic algorithms (GAs) and evolutionary algorithms (EAs) comprise the field of *evolutionary computation* (Fogel 1994a). "Evolutionary algorithms emphasize phenotypic adaptation, while genetic algorithms emphasize genotypic transformations" (Fogel 1993, p. 23). The GA views the evolutionary process as the development and combining of useful schemata, while the EA views the same process as the ability of the individual and the species to adapt to, and to survive a changing environment (Fogel 1993).

The genetic algorithm focusses on the genotype and is typified by the use of crossover as the primary operator to explore the genome space. Point (bit) mutation plays a secondary role. Even a small change in a genotype can result in a big difference in performance.

While genomes are bitstrings in the computer, the evolutionary algorithm focusses on the phenotype and its behavior. Mutation of the structure is the primary operator used to explore the genome space. For example a real number is mutated by adding a Gaussian random variable with a mean of zero (Fogel 1993). Mutation of a finite state machine (FSM) consists of adding or deleting a state; or changing an output, a transition, or the start state (Fogel 1994a). The evolutionary algorithm is based on mutations which aim for small behavioral changes, and has parents and children competing against each other for a place in the following generation (Fogel

1993).

Within the evolutionary algorithm, a *feature* consists of the functional parts of a genome contributing to an individual's behavior. When focussing on the phenotype, a feature is more relevant than the individual bits which more successful individuals have in common. Since crossover is generally not used in an evolutionary algorithm, the physical location of a feature in the genome is not relevant. Let F represent a given feature, and $E(F,g)$ and $N(F,g)$ represent the expected number and actual number, respectively, of individuals which exhibit feature F in generation g . Then following a process similar to that used to develop equation 2.2, for features the equivalent of equation 2.2 is

$$E(F, g+1) = n (1-p_d) N(F,g) p_{AV} \quad (\text{Equation 2.3})$$

where n is the population size, p_d is the probability that feature F will be disrupted (lost due to mutation), and p_{AV} is the average probability that an individual with feature F will be selected to reproduce into the next generation. When $n > \frac{1}{p_{AV}(1-p_d)}$, the number of representatives of feature F can be expected to increase in the next generation (Angeline and Pollack 1993).

As indicated earlier a GA is viewed as manipulating schemata, while an EA aims to modify integrated features. Angeline and Pollack (1993) created a new operator which is based on the theoretical view of features as the most relevant contributing factors in the success of the EA as a search procedure. Angeline and Pollack's compress operator "freezes" a physical part of a genome. It does not freeze bits within a field (functional unit) of the phenotype; only a complete field of a genome is frozen such as a future state or output of an FSM. Theoretically, a

genetic algorithm retains relevant physical units in the form of schemata. But schemata consist of bits, and the functional units of the phenotype are ignored by both schemata and the GA's operators. Mutation alters a single bit within a field, and there is generally nothing to prevent a crossover point from falling within a field.

Another philosophical difference between the GA and EA relates to fitness definition. When a genetic algorithm uses a fitness based selection process and the fitness is an explicitly defined function, the environment consists of a static fitness landscape of valleys and hills for the population to overcome; that is the fitness landscape remains unchanged throughout a run. The population could gravitate towards a local rather than a global optimum when the fitness landscape consists of multiple peaks.

In an evolutionary algorithm however, fitness is determined by competition between the individuals in a population, even when the fitness can be explicitly defined. Each individual competes against a fixed number of randomly chosen members of the population. The population is then ranked based on the number of wins. Generally the selection process retains the top half of the population for the next generation. The remaining half of the next generation is filled by applying mutation to each retained individual, with each individual producing a single child. Consequently, for each succeeding generation parents and children compete against each other for a place in the following generation. This provides a changing environment (fitness landscape) and is less likely to converge prematurely (Fogel 1994a).

The two different philosophies have relative strengths and weaknesses. Each

is more appropriate for different types of problems (Angeline and Pollack 1993), but a competition can easily be integrated into a genetic algorithm to reduce the chance of premature convergence.

The format of the genome used by Jefferson et al. (1992) for breeding a trail following FSM ant, together with Angeline and Pollack's (1993) success with their innovative operators prompted consideration of creating new operators to enhance schemata formation in the GA described in chapters 4 and 5. Since the newly designed operators increase the homogeneity of the genome population, they could make the GA more prone to premature convergence. Competition offers a way to counteract this. These seeds of thought formed the basis for the experimentation which is presented starting in chapter 3.

This section has focussed on the differences between the GA and EA. Common to both are the criteria for termination as discussed in the next section.

2.2.8 Variations on termination

There are different ways to terminate a genetic algorithm. The GA can loop until an individual is found which successfully completes a task. This could result in an infinite loop if the algorithm converges prematurely, or if the problem does not have a solution in the genome space. The genetic algorithm can be stopped after a preselected number of generations have been bred as was the case in the examples developed in sections 2.1 and 2.2.5, or after a preselected number of children have been "born" (Koza 1992). This could halt the evolutionary process just before a solution would have been found. Similarly, when the purpose of an experiment is to

observe the evolutionary process and how the individuals adapt to their environment, as in *Tierra* (Ray 1992), it is important to avoid stopping the process during an interesting evolutionary occurrence. In this case the genetic algorithm can be run as an infinite loop which must be terminated externally. Regardless of the purpose of the experimentation, a looping structure must be implemented to cycle through the generations, and the manner of termination is the one which will best serve the project at hand.

2.3 Reflections on the genetic algorithm

When designing a genetic algorithm for a particular application, the first step is to define the phenotype along with the phenotype-genotype correlation. Then the fitness function and operators must be determined, and the parameters (like population size and mutation rate) given values. After a few runs, the system may be modified to alter its performance, for example by changing the mutation rate or population size. When a genetic algorithm seems to be progressing nicely but is unable to find a solution in the allotted number of generations, the termination criteria can be changed by increasing the number of generations which are bred. When this situation occurs, this simple modification can greatly alter the ability of the GA to succeed. For more complex convergence problems, a new operator may be defined and added to the program, or an old operator may be defined differently. The fitness function may be redefined. This is sometimes done when the reason for the experiments is to determine how differences in an environment can affect the evolutionary process. These are just a few of the ways that a genetic algorithm can

be altered to permit it to explore the genome space when looking for a solution to a given problem or when studying the evolutionary process.

A wide variety of strategies have been reviewed in this chapter. For the research which is presented in this document, the benchmark case is achieved by implementing the work of Jefferson et al. (1992) with different population sizes. Their work is fully described in chapter 4. This implementation, used as the benchmark, is altered in steps, initially by separately adding two newly designed operators which reorganize FSM genomes during run time. Based on schema theory, each of these algorithms could result in either faster convergence to a solution, or premature convergence. Competition is then added to the system to determine its effect on the search process. After determining the effect of these modifications on the efficiency of the genetic algorithm for this one problem, the same experiments are applied to different problems to determine whether the expected improved effectiveness of these algorithms occurs consistently across a range of problems.

>>>●<<<

Chapter 3

Project aims and considerations

The research reported in this thesis examines the effect of the dynamic reorganization of a finite state machine genome on GA efficiency. Dynamic reorganization means that the reorganization of the genome occurs while the algorithm is running. In this context, efficiency has several meanings: either processing fewer individuals to get a solution, and/or using less processor time to get a solution. In addition discovering a smaller FSM for a solution is considered advantageous because smaller machines allow for more efficient design. Angeline and Pollack (1992) found that larger machines tended to include redundancies.

Operators were designed which reorganize the FSM during GA execution (chapter 5), and their effect on the efficiency of the GA are explored (chapters 7 through 9). The nature of the finite state machine permits many representations of equivalent machines simply by changing the names of the states and/or by minimizing the FSM. The new operators (Standardize Future State and Move To Front) are FSM specific and find an equivalent finite state machine in the genome space. These new operators rearrange the FSMs to make equivalent FSMs with the same number of states appear the same so that useful schemata are evident. By reducing equivalent FSMs with the same number of states to a single FSM, equivalent

machines do not have to compete with each other for their share of the population during the selection process. But a serious concern before embarking on the research was whether altering equivalent FSMs to a single representative of their equivalence class might slow down evolution and/or result in premature convergence. The reduction of variety in the population, while enhancing schema growth, reduces innovation from crossover. In addition, the single representative, which is chosen from an equivalence class, might be further removed from a more successful individual than one of the other members in the equivalence class, possibly resulting in a single suboptimal individual taking over the population.

Selection generally results in the reproduction of the most-fit individuals. Crossover generally mixes up what exists. If no bitstring has a 1 in the first position, crossover cannot produce a 1 in that position. Mutation however is innovative. It can create something which didn't exist before, like a 1 in the first position when all individuals have a 0 there. Crossover, depending on how it is defined, might or might not be innovative. When crossover is permitted anywhere along the bitstring, crossover can be innovative. For example, if the last bit of state $00101_2 = 5_{10}$ is crossed with the last bit of state $11000_2 = 24_{10}$, the two new states are $00100_2 = 4_{10}$ and $11001_2 = 25_{10}$. That is crossover has created two states which may not have existed before as next states, thus tapping into an unused part of the genome. If crossover is not permitted within fields this innovative ability of crossover is lost.

In the first sequence of experiments, the new operators were applied to each newly formed individual during each generation. Later experiments applied one of the new operators every i^{th} generation, $2 \leq i \leq 5$. Competition was added to reduce

the chance of premature convergence.

The initial case was modelled after the work done by Jefferson et al. (1992). To facilitate experimentation, the GA was designed to be easily modified for the planned experiments. Mutation inverted bits. Crossover was carried out with a given percent per bit. A mating pair was selected, with one of the two chosen as the starting parent. Crossover was carried out as follows:

```

for i = 1 to genome_length
  if random( ) < probability_of_crossover
    then change to other parent
  child(i) = present_parent(i)

```

where i moves the process along the bits of the genome. In section 4.2, schema theory is reconsidered within the context of this crossover algorithm.

Angeline and Pollack (1993) emphasized the phenotypic properties of the FSM and focussed on the digraph structure of the genome. Using this interpretation, they did not use crossover—only mutation with respect to a next state or output was implemented. Mutation was not carried out on bits, but rather on fields. They used this view, along with competition for selection, as their starting point, and did not compare this to the methods used by Jefferson et al. (1992)

Angeline and Pollack (1993), and Jefferson et al. (1992) represent two similar, but different, approaches to modelling the evolutionary process. The GA, used by Jefferson et al. (1992) focusses on the genotype, with most variation resulting from crossover. Mutation plays a smaller role in the process. The evolutionary algorithm (EA) used by Angeline and Pollack (1993) places more importance on the phenotype and depends solely on mutation for evolution and exploration of the genome space.

Fogel (1991, p. 234) wrote that "When the evolving coding structure [that is a genome] is a finite state machine, the 'meaning' of each state is highly dependent on its relationship with all of the other states in the machine . . . Mutations that would simply substitute states from one machine with those of another are unsuitable and do not adequately reflect the effects of sexual recombination in this case." But Jefferson et al. (1992) point out how different parts of the successful FSM found by their GA seem to be useful for different parts of the trail: essentially some parts of some FSMs are like subroutines which might be useful when combined with part of another genome as a result of crossover. Since crossover is generally not used in the EA, the location of the data along the genome is irrelevant. For these and other reasons the new operators (SFS and MTF) were tested within a GA.

The focus of the research described in this document was to explore strategies which might make a more efficient GA when the population consists of FSMs. A primary aim of the project was to determine to what extent the placement of the FSM within the genome influences the efficiency of a GA. Strategies which process fewer individuals to get a solution, and/or use less processor time, and/or get FSMs with fewer states are preferable. Experimentation was carried out to determine whether these objectives could be attained at the same time. A concern was whether the new strategies would reduce exploration of the genome space thus hampering rather than enhancing the evolutionary process. The evaluation procedures are described in chapter 7, and the experimentation and evaluation of data are covered in chapter 8. The next chapter discusses the work of Jefferson et al. (1992) which was used as the benchmark or starting point, and the programming of the

benchmark.

>>>●<<<

Chapter 4

The benchmark

A system was developed to implement the work done by Jefferson et al. (1992). This system defined an artificial ant as a finite state machine, and bred them to be able to follow a given trail representing a dissipating pheromone (chemical) trail. In approaching the trail-following problem, the importance of the organization of the genome to the convergence of a genetic algorithm can be utilized to try to improve performance. This can be particularly useful because the finite state machine can be used to model the solution to different types of problems, and consequently can be used as the operand of a genetic algorithm searching for such solutions. The next section discusses the finite state machine as a genome within the context of Jefferson et al. (1992)

4.1 The finite state machine as a genome

A finite state machine (FSM) is a transducer. It is defined as an ordered septuple $(Q, s, F, I, O, \delta, \lambda)$, where Q is a finite set of states; $s \in Q$ is the start state; $F \subseteq Q$ is the set of final states; I is a finite set of input symbols; O is a set of output symbols; $\delta: Q \times I \rightarrow Q$ is a transition function; and $\lambda: Q \times I \rightarrow O$ is an output function. A finite state machine is initially in state s . It receives as input a string of symbols.

An FSM which is in state $q \in Q$ and receiving input symbol $a \in I$ will move to state $q_{\text{next}} \in Q$ and produce output $b \in O$ based on transmission rule $\delta(q,a)=q_{\text{next}}$ and on output rule $\lambda(q,a)=b$. This data structure has been used to study diverse problems in conjunction with a simulated evolutionary process. Angeline (1994), Fogel (1991), and Stanley et al. (1994) used FSMs to explore the iterated prisoner's dilemma. MacLennan (1992) represented simulated organisms as FSMs to study communication development. And, Jefferson et al. (1992), and Angeline and Pollack (1993) used a finite state machine genome to breed a trail-following strategy.

The genome used by Jefferson et al. (1992) had 453 bits (See figure 4.6.) and therefore defined a space of 2^{453} bitstrings. This space can be divided into equivalence classes with each class containing equivalent FSMs. For example, the two machines in figure 1.1 are the same machine: q_1 and q_2 have simply exchanged names. The genome used by Jefferson et al. did not utilize this fact to build and retain useful schemata. Since there are many equivalent FSMs, a number of variations of a nearly successful FSM may exist in a population, but the similarity may not be evident due to different representations. Useful schemata are competing with each other during the selection process, each hampering the growth of the other.

Theoretically, the genetic algorithm moves a population towards a solution to a problem as useful schemata grow in number and length. Because a single finite state machine has different representations (simply by changing state names), identical machines with different representations do not necessarily share a useful schema; that is a set of identical transitions/outputs which provide a successful strategy could reside in different parts of the genome, and appear very different due

to having different state names. Consequently, these machines hamper schema growth as the machines compete against each other for their share of the next generation. The reorganization algorithms presented in chapter 5 compensate for this. The new operators, called SFS (standardize future/next state) and MTF (move to front), will reorganize finite state machines during run time so equivalent machines with the same number of states will appear the same. The purpose of these new operators is to enhance the growth of useful schemata and thereby hasten the convergence of the genetic algorithm. The strategies these operators employ are FSM-specific, and theoretically should improve the convergence of the genetic algorithm for FSM genomes.

The benchmark or starting point for the research presented in this document is based on the work of Jefferson et al. (1992) The benchmark is presented in the next section along with its implementation. In chapter 5, the reorganization algorithms will be added to this benchmark.

4.2 The benchmark: description and theory

The benchmark is described in this section. Equations 2.2 and 2.3 (sections 2.4 and 2.2.7 respectively) are redeveloped to apply to the variations on the simple genetic algorithm which are implemented by Jefferson et al. (1992) The benchmark for this study implemented the work done by Jefferson et al. They defined an artificial ant as a finite state machine, and bred them to be able to follow a given trail representing a dissipating pheromone trail. The trail has "a series of turns, gaps, and jumps that get more difficult as [they] progress" (Jefferson et al. 1992, p. 555).

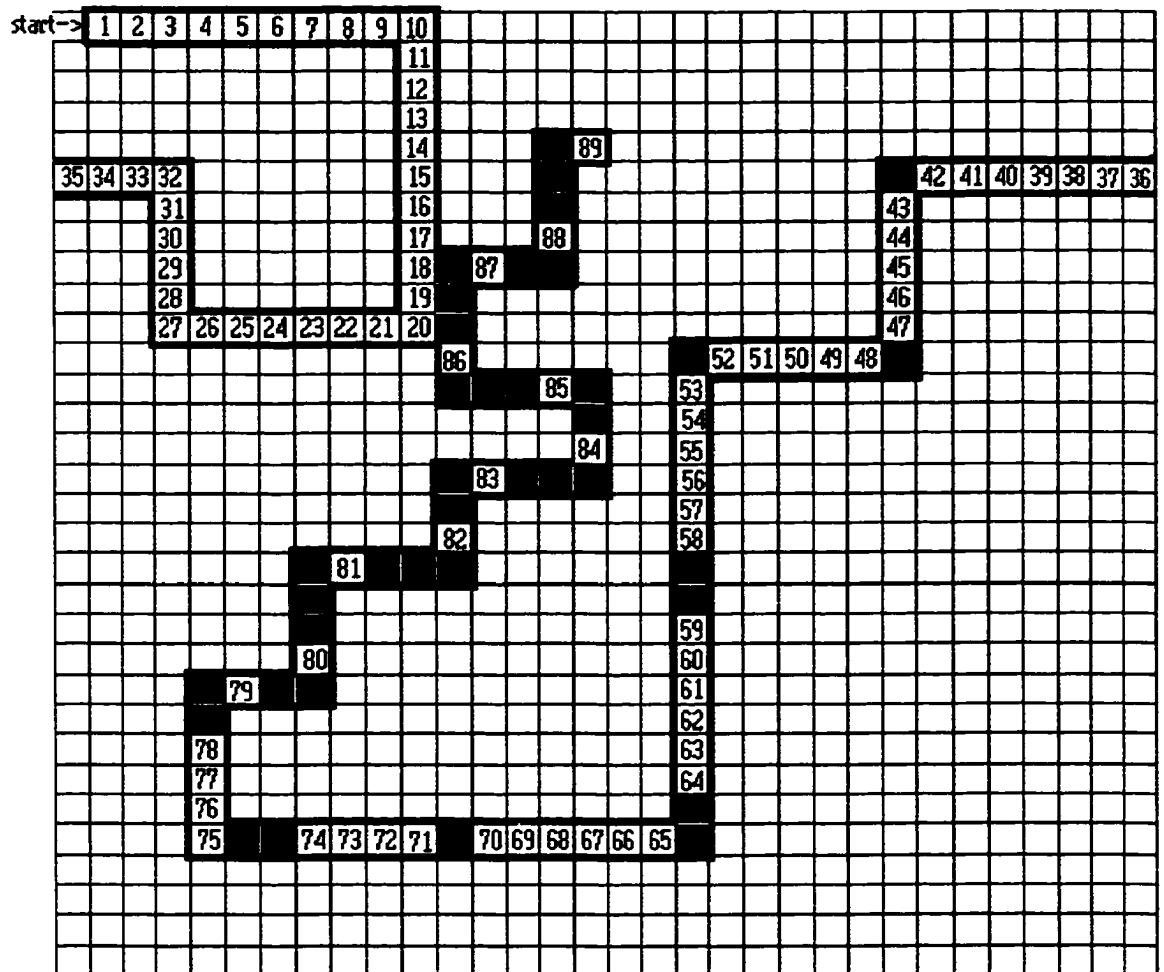


Figure 4.1: The John Muir Trail (Printed with permission of Perseus Books, LLC)

While Jefferson et al. (1992) report results on the John Muir Trail (figure 4.1), Koza (1992) used the Santa Fe Trail (figure 4.2). The John Muir Trail starts off with three straightaways, each followed by a right turn. Jefferson et al. used this trail to see if the ants would develop a bias for right turns. Koza (1992) suggests that the Santa Fe Trail is the more difficult of the two. In the diagrams of the trails, the numbered squares represent the marked portions of the trails, that is the part of the

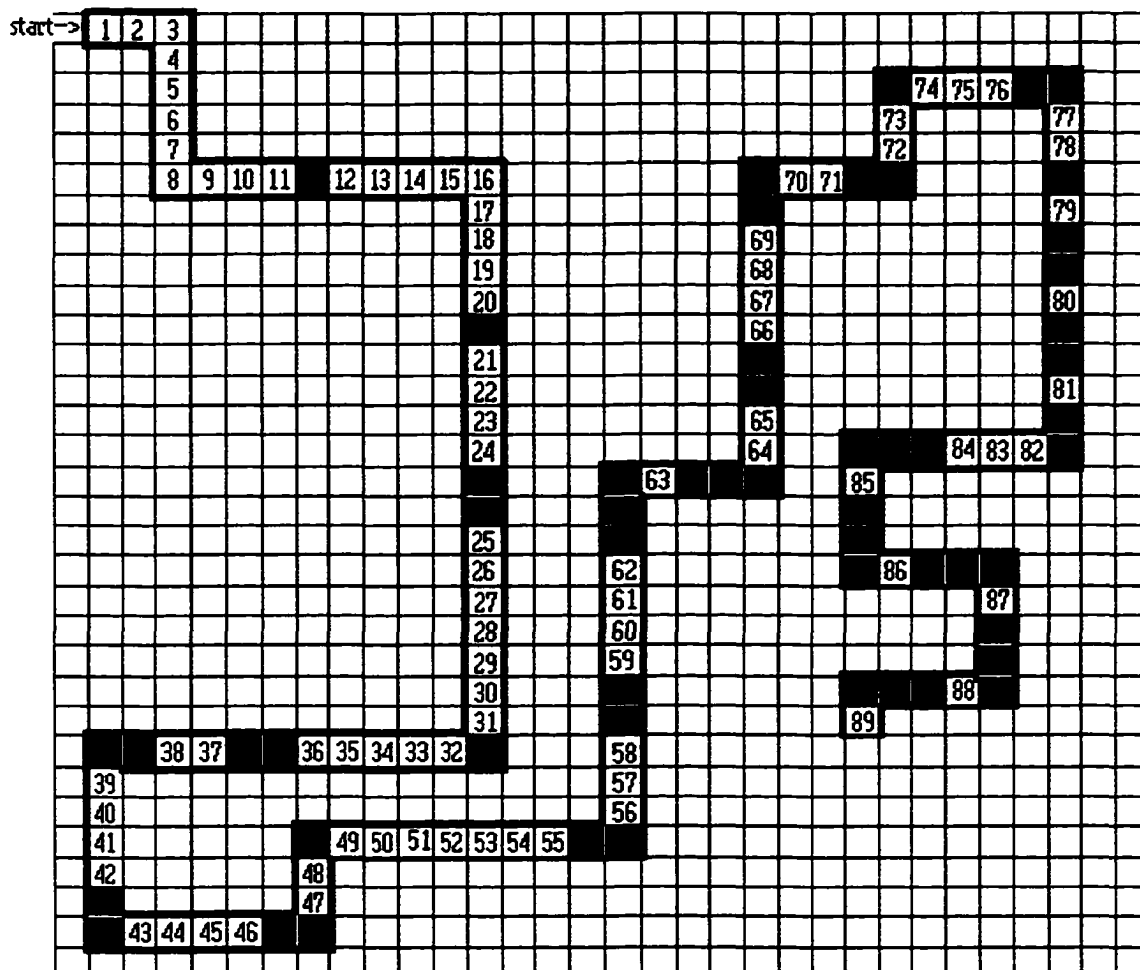


Figure 4.2: The Santa Fe Trail (Printed with permission of Perseus Books, LLC)

trails which still have pheromone. The black squares represent the part of the trails from which the pheromone has evaporated and appear the same as the background (non-trail) squares to the ant (figure 4.3). Each trail lies on a 32×32 toroidal grid, and each trail consists of 89 marked steps. In addition to the 89 marked steps, the John Muir trail has 38 steps which have lost the pheromone and must be interpolated

an ant can sense the presence/(1) or absence/(0) of pheromone in the box directly ahead; this is the input to the FSM. The output of the FSM directs the ant's action for the given time step. The ant can either turn 90° to the left or right, move 1 step ahead, or do nothing; the latter permits the ant to change its state without taking any action. Ants are permitted 200 time steps to traverse the trail. The fitness consists of the number of marked trail steps an ant is able to cover in the 200 time steps; consequently doing nothing just wastes a time step. Jefferson et al. (1992) found that the best FSMs always directed the ant to take some action (turn in place or move ahead) when a transition rule was applied; that is, some FSM minimization was occurring because the fitness function encouraged this. To prevent retracing part of the trail or back-tracking on the trail, when an ant steps on a marked step the marking is erased. No additional credit is given for completing the trail in fewer than 200 time steps.

It seems that a good trail following strategy should move the ant forward when there is a marked step directly in front. When there is no trail ahead, look to the right and to the left to see if the trail has turned; if the trail is gone, continue in the same direction as before. Since the John Muir Trail contains 12 right turns and only 8 left turns, it seems that, when there is no trail ahead, a right turn would be a good first step. If after the turn, the trail is still not in evidence, an about face (2 left turns or 2 right turns) is appropriate. If there is still no trail ahead, turn back to the original direction (a right turn) and then step forward. This strategy makes the solution to the problem seem simple, but the imposition of a limit of 200 time steps makes it difficult. The FSM in figure 4.4 (similar to one presented by Jefferson et

al. 1992) permits the ant to follow the trail, but it only covers 81 trail steps in 200 time steps. The remainder of the trail takes a significant number of additional time steps since there are many unmarked boxes on the remainder of the trail, and it takes 5 time steps to move from the present position to an unmarked box which immediately follows. This FSM ant needs 314 time steps to get a perfect score of 89.

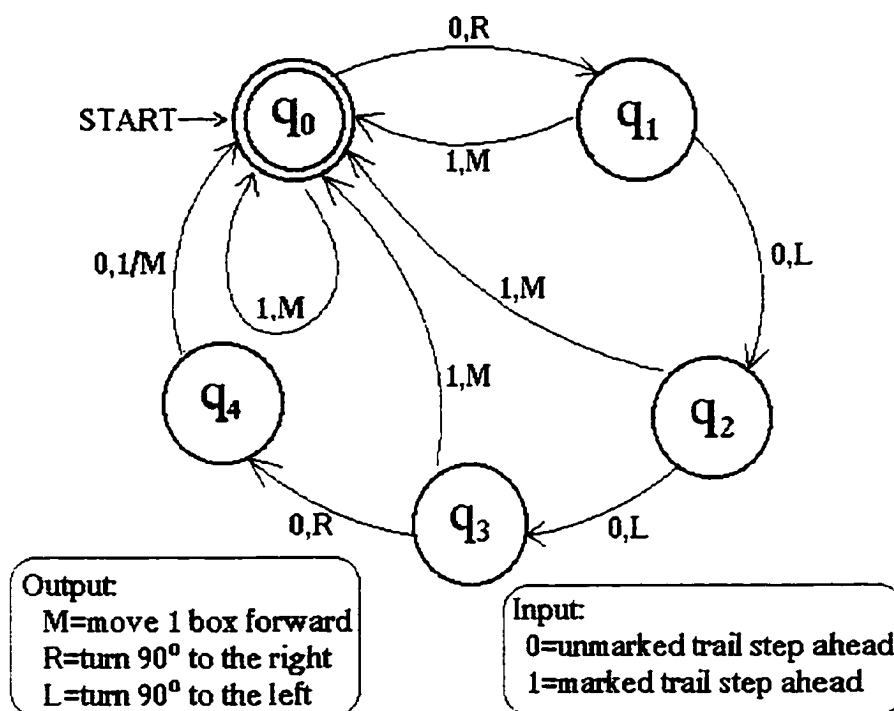


Figure 4.4: Strategy favoring right turns when the next step is unmarked

The sequence of turns needed for the John Muir Trail is

RRLLRLRRRLRLRLLRRLR.

This has five right turns followed by another right turn, two left turns followed by another left turn, and twelve times a turn in one direction is followed by a turn in the

opposite direction. Therefore it is more likely that an ant will have to turn in the opposite direction next time the trail turns. Altering the FSM of figure 4.4 to alternate between right and left turns when the trail ceases to exist in the step directly ahead (figure 4.5) also covers only 81 trail steps in 200 time steps, and it needs 313 time steps to complete the trail. For all practical purposes, nothing has been gained timewise, further indicating that the imposition of a time limitation is what makes this a challenging problem.

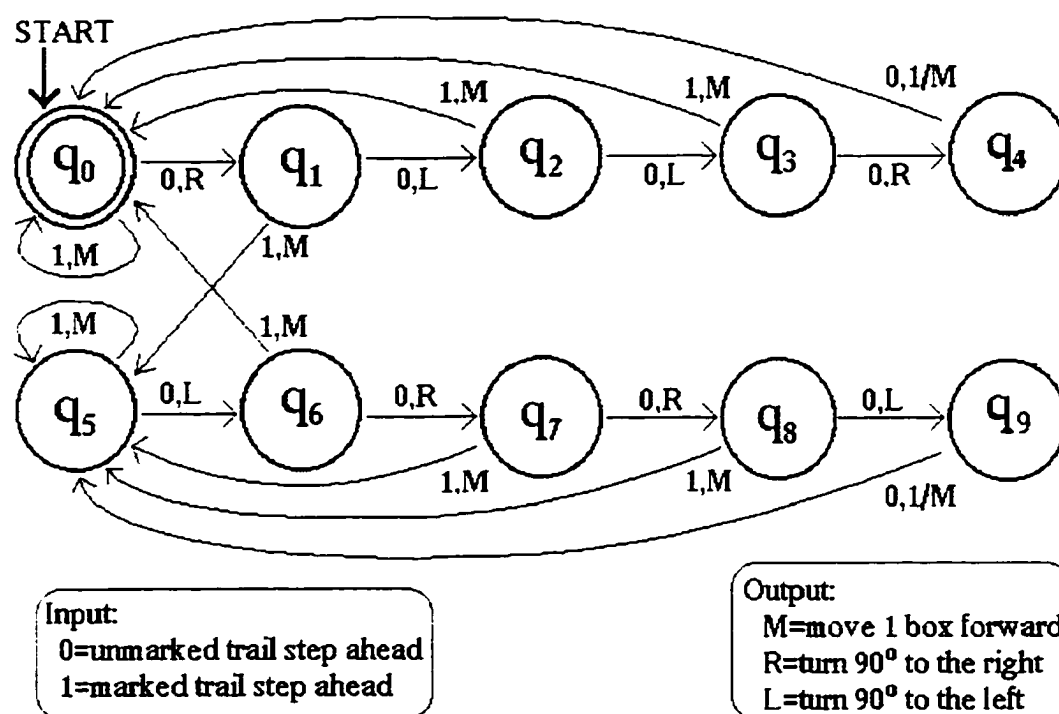


Figure 4.5: Strategy alternates left and right turns when next step is unmarked.

In order to determine the difficulty of finding a successful ant in the defined genome space, Jefferson et al. (1992) randomly generated 1,349,517,312 FSMs and

found the fitness of each. Slightly over 40% of those generated had a score of zero. Slightly over 20% covered only the first marked trail step in 200 time steps, and just under 10% had a score of two. The general trend was fewer FSMs for higher scores, with occasional anomalies occurring. Only 64 of the 1,349,517,312 randomly generated ants covered 80 trail steps in the allotted time, and 10 scored 81. None of the randomly generated FSMs scored above 81.

The FSM genome used by Jefferson et al. (1992) allows for a maximum of 32 states. Each FSM consists of a table with 32 lines for states q_0 - q_{31} , and has two columns for inputs: 0 (no pheromone on the step immediately in front of the ant) and 1 (pheromone present on that next step). The genome contains the next state/output recorded along each row and the sequential listing of the rows of the FSM table. In addition the initial state can be any one of the 32 states. Since there are 32 states, 5 bits are needed for the state number. Four possible outputs (turn left or right, move ahead, do nothing) require 2 bits.

Each ant (FSM) is represented by a genome consisting of 453 bits (figure 4.6) which was initialized by randomly setting all 453 bits. The first 5 bits of the genome indicate the start state. The next $64(5+2)=448$ bits represent the next state (5 bits) and output (2 bits) for q_0 with input 0, q_0 with input 1, q_1 with input 0, q_1 with input 1, and similarly down the table (Jefferson et al. 1992). Consequently, the position of the future state/output for a given present state/input is fixed on the bitstring. While the genome allows for 32 states, the actual FSM which any one genome represents may have fewer than 32 states. For example the FSM with start state q_{13} and state transition table 4.1 uses only 4 states. Other parts of the genome

may eventually be used as a result of crossover or mutation. For example by inverting a bit within the state portion of the genome, mutation could change the next state for q_{13} with input 0 from q_5 ($5_{10}=00101_2$) to q_4 ($4_{10}=00100_2$) or q_7 ($7_{10}=00111_2$), amongst others.

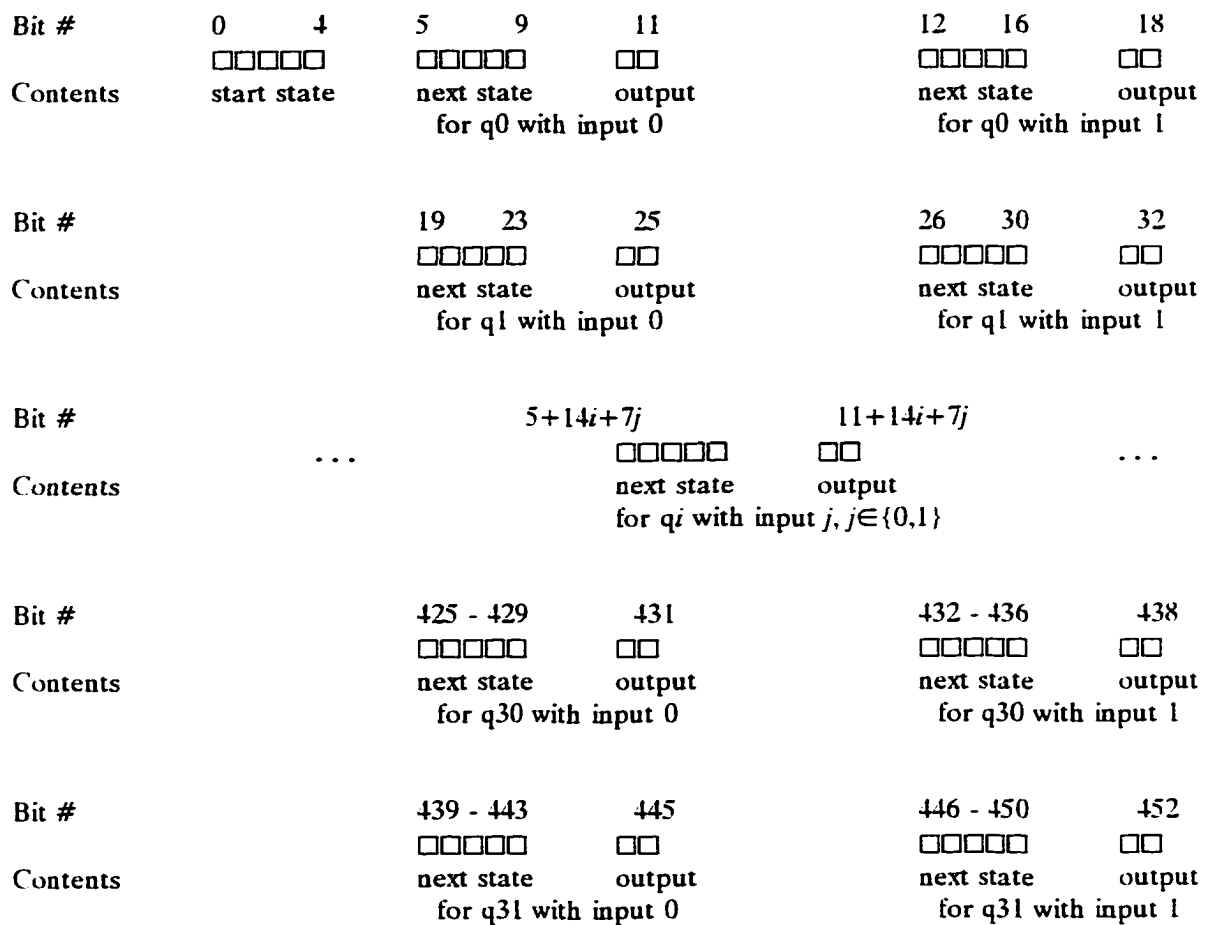


Figure 4.6: 32-state/453-bit FSM Genome Map

Start state: q13

present state	next state/output for input=0	next state/output for input=1
q13	q5/0	q9/2
q5	q13/2	q5/2
q9	q5/1	q24/2
q24	q13/3	q24/2

Table 4.1: Four State FSM

The population of Jefferson et al. (1992) consisted of 65,536 (64K) FSMs. The bottom 95% (least-fit) of each generation was discarded. 65,536 mating pairs were selected from the remaining 5% of the population without regard to fitness. Crossover was applied with a probability of 1% per bit: that is a random number between 0 and 1 was generated for each bit along the genome, and when the number was under 0.01 the subsequent genetic material was taken from the other parent. (Note that in GAs, genetic material is frequently exchanged between two parents producing two offspring. However Jefferson et al. retained only one of the children.) A 1% per bit mutation rate was used, with mutation inverting a bit. Using these parameters and operations, Jefferson et al. got a perfect scoring ant in generation 52.

To find the equivalent of equations 2.2 and 2.3 (sections 2.4 and 2.2.7 respectively) for this scenario, let $E(S, g+1)$ be the expected number of individuals which match schema S in generation $g+1$, and $T(S, g)$ equals the number of individuals which match schema S from the top 100t% (based on fitness) of generation g . Note that t is in decimal rather than percent form. (For example, for

the Jefferson et al. (1992) case mentioned in the previous paragraph $t=0.05$.) Then p_s , the probability that an individual with schema S is selected as a parent, is $\frac{T(S,g)}{nt}$, where n is the population size. Then the probability that schema S is not represented in a mating pair is $\left(1 - \frac{T(S,g)}{nt}\right)^2$, assuming sampling with replacement. Therefore the probability that schema S is present in any given mating pair is

$$1 - \left(1 - \frac{T(S,g)}{nt}\right)^2 = 2 \frac{T(S,g)}{nt} - \left(\frac{T(S,g)}{nt}\right)^2 = \frac{T(S,g)}{nt} \left(2 - \frac{T(S,g)}{nt}\right) \geq \frac{T(S,g)}{nt}.$$

The inequality reflects the fact that $T(S,g) \leq nt$, and as $T(S,g) \rightarrow nt$, $\frac{T(S,g)}{nt} \rightarrow 1$ as its maximum value. Note that when $T(S,g) = nt$, every member of the parent pool matches schema S . Since the probability that at least one parent will match S $\geq \frac{T(S,g)}{nt}$, when the parents are selected to fill the n members of the next generation

$$E(S, g+1) \geq n \frac{T(S,g)}{nt} = \frac{T(S,g)}{t} \quad (\text{Equation 4.1}).$$

But this does not take into consideration the losses of schema S due to crossover and mutation.

Let p_x be the per bit probability of crossover, and p_m be the per bit probability of mutation. Let c be the number of positions in schema S at which crossover can occur (defining length), and b represent the number of fixed bits in the schema. Then the probability that a schema survives crossover is the probability that there is no crossover point within the schema which is $(1-p_x)^c$. Similarly, the probability that S is not lost due to mutation is $(1-p_m)^b$. Hence, the probability that schema S survives both crossover and mutation equals $(1-p_x)^c(1-p_m)^b \geq (1-cp_x)(1-bp_m) \geq (1-cp_x-bp_m)$. Incorporating this into equation 4.1 yields

$$E(S,g+1) \geq \frac{T(S,g)}{t} (1 - cp_X - bp_M) \quad (\text{Equation 4.2}).$$

Thus schema S is most likely to carry into the next generation when it matches a large number of genomes in the parent pool, has a short defining length (c), and fewer fixed bits (b).

Since only the top 100t% of the population serves as parents, S need not be present in a large part of the next generation; it just has to be prevalent in the top 100t% of it. More than $T(S,g)$ representatives can be expected in the next generation when $\frac{1 - cp_X - bp_M}{t} \geq 1$, that is when $cp_X + bp_M \leq 1 - t$. If every member of the parent pool matches S there is no loss due to crossover. For this case equation 4.2 becomes

$$E(S,g+1) \geq \frac{T(S,g)}{t} (1 - bp_M) \quad (\text{Equation 4.3})$$

indicating that schema S will only be disrupted by mutation, and that it is less likely to be lost when there are fewer fixed bits. In this case $T(S,g)$ can be expected to retain the constant value of nt as long as $\frac{1 - bp_M}{t} \geq 1$, or $b \leq \frac{1 - t}{p_M}$.

With a 453-bit genome and $p_X = p_M = .01$, cp_X , bp_M , or $cp_X + bp_M$ can exceed one. In this case the lower bound for $E(S,g+1)$ in equations 4.2 and 4.3 will be negative. This can occur because $1 - cp_X - bp_M$ is used as an approximation for $(1 - p_X)^c (1 - p_M)^b$, and $1 - bp_M$ is used as an approximation for $(1 - p_M)^b$. In such cases, either set the lower bound for $E(S,g+1)$ to zero, or calculate a more accurate lower bound for $E(S,g+1)$ by replacing $(1 - cp_X - bp_M)$ with $(1 - p_X)^c (1 - p_M)^b$ or $(1 - bp_M)$ with $(1 - p_M)^b$ in equations 4.2 and 4.3 respectively. Note that equations 4.2 and 4.3 set lower bounds for $E(S,g+1)$, and the number of individuals in generation g which actually match S will generally be higher than the indicated lower bound.

Jefferson et al.'s work (1992) with the John Muir Trail has been chosen as the benchmark method because the problem has been widely explored. In addition Angeline and Pollack (1993) used it to test the effect of their innovative operators on the convergence of a simulated evolutionary process. This is the aim of the research with the two reorganization algorithms (described in chapter 5). It is important to note that Jefferson et al. were interested in simulating a natural evolutionary process (although differences can be found when a GA is compared to the biological process (Fogel 1993)), while the present research is concerned with producing a search procedure which converges to a solution "faster" and which locates smaller machines. Relevant to this goal is the fact that in nature, a set of genes that defines a specific characteristic (for example eye color) has a set location in a genome, but the nature of Jefferson et al.'s FSM genome defies this natural phenomenon. The start state alone can be in any one of 32 different positions.

As indicated earlier identical finite state machines need not appear the same. For example the two machines in figure 1.1 consist of states q_0 , q_1 , and q_2 . This machine put into the 32 state genome shown in figure 4.6 can have any one of $32 \times 31 \times 30 = 29,760$ different representations. Many of these machines don't even have state names in common, nor need they have the same start state when they do have the same set of state names. Schema growth is not encouraged by this genome since equivalent states of equivalent machines (differing only in state names) do not necessarily reside in the same location of the genome. It is this realization which prompted the design of the two reorganization algorithms which are the focus of this research, and which are incorporated into the benchmark (chapter 5). In the next

section the benchmark program is detailed.

4.3 The coding

The benchmark program retains the selection process, crossover and mutation rates and algorithms, and phenotype-genotype mapping, utilized by Jefferson et al. (1992) as described in the previous section. The initial population is created using Goldberg's (1989) random number generator to create 57 bytes (with values of 0 to 255) per genome, as opposed to 453 bits per genome. Since Jefferson et al. reported that their GA found a successful ant in generation 52, our GA was permitted to breed a maximum of 70 generations when using the same population size.

The programs were coded in C and were executed on a variety of Sun Sparc Stations. The program was designed so that planned modifications could be easily incorporated. To permit additional future testing on other types of problems. parameters which are constant for this problem are given values with `#define` statements.

First the run number and method, and files to receive the output are defined as follows:

```
/*          (r)un number (1) for (b)enchmark          */
#define run_name "Base case modelled after Jefferson et al."
#define run_num 1                                     /* run number */
#define printed_output "rlb.st"                       /* statistics output */
#define perfect_score "rlb.ps"                         /* perfect score file */
```

The file with the `st` extension receives the statistics for the run, and the file with the `ps` extension receives the tables for the successful ants which are bred. In both file names the `r` precedes the run number and the `b` indicates that the run is for the

benchmark. These lines are to be appropriately changed for each run, or the program can be modified to read in these values.

Then the FSM and GA parameters are defined.

```
#define bits_per_state_num 5
#define max_num_states 32 /* 2^bits_per_state_num */
#define num_of_inputs 2
#define bits_per_output 2

/*****
/*   Genome consists of start state followed by sequential listing   */
/*   of the rows of the transition/output table                       */
/*                                                                    */
/*   bits_per_genome = bits_per_state_num +                          */
/*                   num_of_inputs * max_num_states *                */
/*                   (bits_per_state_num + bits_per_output);         */
*****/

#define bits_per_genome 453
#define bytes_per_genome 57 /* contains bits_per_genome bits */

/*****
/*                                                                    */
/*                   Genetic Algorithm Constants                      */
/*                                                                    */
*****/

#define popsize 65536 /* population size */
#define select 3277 /* choose parents from top members of population */
#define maxfitness 89 /* maximum fitness which can be attained */
#define last_generation 70
#define mutation_rate .01
#define xover_rate .01
```

The data types needed for the program are defined as follows:

```
typedef int table[max_num_states][num_of_inputs]; /* transition table */
typedef unsigned char byte;
typedef byte unpacked_genome [bits_per_genome];
typedef byte packed_genome [bytes_per_genome];
typedef packed_genome population [popsize];
```

Two tables are needed for an ant. One table will hold the next states for each present state/input pair, and the other similarly stores the outputs. The first index into each table is the present state number. The second is the value of the input.

With a population of 64K FSMs efficient storage of the population is a concern, so a generation of FSMs is stored in a bytestring format which is the data

type `packed_genome`. `population` holds a complete generation of ants. When converting a bytestring to a table, or vice versa, bits of the genome are manipulated. Consequently, for these conversions an intermediate form of 453 bits is employed. Hence, the genome is stored as a bitstring, that is as data type `unpacked_genome`, for the conversion.

Finally the variables are declared.

```
population pop[2];      /* holds present and newly evolving generations */
table next_state, output;
unpacked_genome genome;

int generation=0,
    old_pop=0, new_pop=1, /*the 2 generations' indices in the pop array*/
    start_state,
    fitness[popsize],
    fitness_count[maxfitness+1]={0}, /* fitness histogram */
    sorted[3][popsize]={0}, /*for merge-sort: indices of population */
    row_of_sorted=1; /* row of sorted array that has result of the sort*/

FILE *fp_stats, *fp_perfect;

double seed[31]={0, .74903538, .39438263, .69343845, .03923732,
                  .57838453, .29652832, .48296342, .13742938,
                  .80347386, .93847534,
                  .61563214, .27384864, .37929661, .41974869,
                  .20203185, .72399187, .32661661, .55001424,
                  .1917149, .82574681,
                  .65795069, .00132096, .44038342, .097113497,
                  .78874573, .51412962, .90991612, .16076033,
                  .99843391, .8582388 };

```

The declaration for the `pop` array doesn't reveal the complexity of the structure. This array actually requires three indices. The first index can be a zero or one. One value, a zero or a one, is stored in `old_pop` and indicates the location of the population of parents. The other value is stored in `new_pop` and indicates the row into which the newborns are to be placed. The values of `old_pop` and `new_pop` are exchanged as each new generation becomes the parents for the subsequent generation (double buffering). But `pop` is of type `population`, which is itself an array. Therefore the second index of `pop` is the number of the individual in the

population. The data type **population** is a **packed_genome**, so the third index of **pop** permits accessing the bytes (and hence the bits) of the genome for the individual pointed to by the first two indices of **pop**.

30 runs are executed for each method. **run_num** is the index into the **seed** array, providing the seed for the random number generator. Consequently each set of 30 runs starts with the same 30 initial populations.

With these parameters and variables defined, the program can be coded. Before discussing the GA proper, there are several utility procedures which are needed. The most complex utilities required are a random number generator and a sort procedure. The random number generator coded in Turbo-Pascal in Goldberg's book (1989) has been converted to C. A stable merge-sort is implemented (Knuth 1973), but any sorting procedure can be used. The population is sorted in order of decreasing fitness. The result of the sort appears in the array **sorted**. The first index of the **sorted** array indicates the row of the array which holds the result of the sort and is given by **row_of_sorted**. The indices of the individuals in the **pop** array are stored in **sorted**. So for example, the most-fit individual is **pop[new_pop][sorted[row_of_sorted][0]]**, and the least fit individual is **pop[new_pop][sorted[row_of_sorted][popsize-1]]**.

```

/* ----- */
/*           Power base 2           */
/* ----- */

int power_of_2 (int exponent)
/* this will only return the proper answer when the exponent is non-neg */

```

Figure 4.7: Calculate 2^n (continued on next page)

```

if (exponent < 0) answer = 0;
else
    for (answer = 1; exponent; --exponent)
        answer *= 2; return answer;
} /* power_of_2 */

```

Figure 4.7 continued: Calculate 2^n

`power_of_2(n)` (figure 4.7) calculates 2^n for a non-negative `n`. If `n < 0`, `pow_of_2` returns a 0. This is used by `main` to test the validity of the `#defined` values.

As stated earlier, the genotype is stored as 57 bytes. It is converted to a table with base 10 values. The conversion from bytestring to table (`bytestring_to_FSM`, figure 4.12), and vice versa (`FSM_to_bytestring`, figure 4.13), requires conversions between binary and base 10 numbers (`bin_to_dec` in figure 4.10, and `dec_to_bin` in figure 4.11). These conversions utilize an intermediate form of the genome, a bitstring with 453 bits. Therefore routines are needed to unpack the 57 bytes into 453 bits (`unpack_genome`, figure 4.9) and pack the 453 bits into 57 bytes (`pack_genome`, figure 4.8).

`pack_genome` (figure 4.8) packs 8 bits into a byte. `i` is the index into the array of 57 bytes, `j` is the index into the bitstring, and `k` limits each byte to 8 bits. Indices `i`, `j`, and `k` serve the same purpose in `unpack_genome` (figure 4.9). `pack_genome` shifts a byte one bit left, and ORs it with the next bit to fill the eight bits of a byte (lines 4-5, figure 4.8). Since 57 bytes=456 bits, the last 3 bits of the last byte are left blank (lines 7-8, figure 4.8). In `unpack_genome` (figure 4.9), the single 1 in `mask` (line 4) is used to isolate the individual bits of a byte. As the `mask`

is shifted and ANDed with a byte the individual bits of the byte are sequentially isolated (lines 4-8).

```

/* ----- */
/*           Pack Genome           */
/* ----- */

void pack_genome(unpacked_genome bitstring, packed_genome bytestring)
{
line #
1:   int i, j, k;
2:   for (i=0, j=0; j<bits_per_genome; ++i) {
3:       bytestring[i] = 0;
4:       for (k=0; k<8 && j<bits_per_genome; ++k, ++j)
5:           bytestring[i] = bytestring[i] << 1 | bitstring[j];
6:   }
7:   if (k < 8)                /* fill in extra bits in last byte */
8:       bytestring[bytes_per_genome-1] <= (8 - k);

} /* pack_genome */

```

Figure 4.8: Bitstring to Bytestring Conversion

```

/* ----- */
/*           Unpack Genome           */
/* ----- */

void unpack_genome(packed_genome bytestring, unpacked_genome bitstring)
{
line #
1:   int i, j, k;
2:   byte mask;
3:   for (i=0, j=0; j < bits_per_genome; ++i)
4:       for (k=0, mask=0x80; k<8 && j<bits_per_genom; ++k, ++j, mask>>=1)
5:           if (bytestring[i] & mask)
6:               bitstring[j] = 1;
7:           else
8:               bitstring[j] = 0;

} /* unpack_genome */

```

Figure 4.9: Bytestring to Bitstring Conversion

bin_to_dec (figure 4.10) converts a binary integer to its decimal equivalent.

The loop (lines 2-3) is based on the binary expansion

$$2^n b_n + 2^{n-1} b_{n-1} + 2^{n-2} b_{n-2} + \dots + 2^i b_i + \dots + 2^2 b_2 + 2^1 b_1 + 2^0 b_0 =$$

$$b_0 + 2(b_1 + 2(b_2 + 2(\dots + 2(b_i + \dots + 2(b_{n-2} + 2(b_{n-1} + 2b_n)) \dots)))$$

or

$$(((\dots(((b_n \times 2) + b_{n-1}) \times 2 + b_{n-2}) \times 2 + \dots + b_i) \times 2 + \dots + b_2) \times 2 + b_1) \times 2 + b_0$$

of the binary integer $b_n b_{n-1} b_{n-2} \dots b_i \dots b_2 b_1 b_0$. The **start_position** is advanced (line 2) as each bit is processed. **num_of_bits** gives the number of bits in the binary number being converted. When **bin_to_dec** returns control to the calling program (line 4), **start_position** indicates the start of the next field in the genome.

```

/* ----- */
/*           Binary to Decimal           */
/* ----- */

int bin_to_dec(int *start_position, int num_of_bits,
               unpacked_genome bitstring)

/* when this function terminates, start_position is the start
   position of the next field to be processed */
{
line #
1:     int i, dec=0;

2:     for (i=0; i<num_of_bits; ++i, ++*start_position)
3:         dec = 2*dec + bitstring[*start_position];
4:     return dec;

} /* bin_to_dec */

```

Figure 4.10: Binary to Decimal Conversion

dec_to_bin (figure 4.11) utilizes the fact that dividing a decimal integer d by two to get an integer quotient and a remainder generates the least significant bit

of its binary equivalent. Repeatedly divide sequentially derived quotients by two and retain the remainders. The remainders are the bits of the binary equivalent. The bits of the binary equivalent of d are generated from least significant to most significant. More formally

$d = \lfloor d/2 \rfloor + b_0$. Let $q_1 = \lfloor d/2 \rfloor$.
 Then $q_1 = \lfloor q_1/2 \rfloor + b_1$. Let $q_2 = \lfloor q_1/2 \rfloor$.
 Then $q_2 = \lfloor q_2/2 \rfloor + b_2$. Let $q_3 = \lfloor q_2/2 \rfloor$.
 $\vdots$
 Then $q_i = \lfloor q_i/2 \rfloor + b_i$. Let $q_{i+1} = \lfloor q_i/2 \rfloor$.
 $\vdots$

This iterative process continues until $q_n = 0$ for some n . Then the binary equivalent of d is $b_n b_{n-1} \cdots b_2 b_1 b_0$.

```

/* ----- */
/*           Decimal to Binary           */
/* ----- */

void dec_to_bin(int *start_position, int num_of_bits, int number,
               unpacked_genome bitstring)

/* when this function terminates, start_position is the start
   position of the next field to be processed */
{
line #
1:     int i;

2:     for (i=num_of_bits-1; i>=0; --i, number /= 2)
3:         bitstring[*start_position + i] = number % 2;

4:     *start_position += num_of_bits;

} /* dec_to_bin */

```

Figure 4.11: Decimal to Binary Conversion

In `dec_to_bin` (figure 4.11), the index i controls the storage of bits into

the **bitstring**. Since this **i** is counted down by the **for** loop (line 2), the most significant bit is stored when **i=0**. Since the 453 bits of the **bitstring** must be appropriately converted into state numbers (0-31) and outputs (0-3), the **start_position** is changed (line 4) to indicate the beginning of the next field to be processed. As in **bin_to_dec**, when **dec_to_bin** returns control to the calling program, **start_position** indicates the start of the next field in the genome.

```

/* ----- */
/*          Store byte string          */
/*          information as a           */
/*          finite state machine       */
/* ----- */

void bytestring_to_FSM(int *start_state, packed_genome bytestring)
{
line # 1:   int i, state, input;
2:   unpacked_genome bitstring;

3:   unpack_genome(bytestring, bitstring);
4:   i = 0; /* present position in bitstring */
5:   *start_state = bin_to_dec(&i, bits_per_state_num, bitstring);

6:   for (state=0; i<bits_per_genome; ++state)
7:     for (input=0; input<num_of_inputs; ++input) {
8:       next_state[state][input] =
9:         bin_to_dec(&i, bits_per_state_num, bitstring);
10:      output [state] [input] =
11:        bin_to_dec(&i, bits_per_output, bitstring);
12:    }

} /* bytestring_to_FSM */

```

Figure 4.12: Bytestring to Table Conversion

bytestring_to_FSM (figure 4.12) and **FSM_to_bytestring** (figure 4.13) invoke the four previously described subroutines. For **bytestring** to table conversion, **bytestring_to_FSM** first unpacks the 453 bits out from the 57 bytes (line 3, figure

4.12). The `start_state` is then decoded (lines 4-5). Then the `for` loops (lines 6-7) march through the 32 states of the FSM and the two inputs for each state to fill in the `next_state` and `output` tables (lines 8-11).

`FSM_to_bytestring` (figure 4.13) is needed to reverse the process when the table has been altered. First each functional part of the FSM is converted to its binary form. The bits are placed in their proper position in the bitstring as shown in figure 4.6. As in `bytesting_to_FSM`, conversion starts with the `start_state` (lines 3-4, figure 4.13). The `for` loops (lines 5-6) move through the 32 states and the two inputs for each state. Lastly the 453 bits are packed into 57 bytes for more efficient storage (line 11).

```

/* ----- */
/*      Store finite state      */
/*      machine information     */
/*      as a byte string        */
/* ----- */

void FSM_to_bytestring(int start_state, packed_genome bytestring)
{
line # 1:   int i, j, k;
2:   unpacked_genome bitstring;

3:   k = 0; /* present position in bitstring */
4:   dec_to_bin(&k, bits_per_state_num, start_state, bitstring);

5:   for (i=0; i<max_num_states; ++i)
6:     for (j=0; j<num_of_inputs; ++j) {
7:       dec_to_bin(&k, bits_per_state_num,
8:                 next_state[i][j], bitstring);
9:       dec_to_bin(&k, bits_per_output, output[i][j], bitstring);
10:    }
11:    pack_genome(bitstring, bytestring);

} /* FSM_to_bytestring */

```

Figure 4.13: Table to Bytestring Conversion

With the utilities complete, the program for the GA itself can be written. The fitness function, `calc_fitness` (figure 4.14), is problem specific. The `trail` array contains the trail, listed row by row from figures 4.1 and 4.3. The ones in the array indicate the marked trail steps. When an ant covers a marked step, the marking is erased to prevent the ant from backtracking on the trail. To avoid loss of the `trail`, it is local to `calc_fitness` and is therefore set up each time the subroutine is invoked.

When `calc_fitness` (figure 4.14) is invoked, the ant is positioned in the upper left box of the trail grid (line 1) and is facing the first step of the trail (line 3). `i` is the row number and `j`, the column number of the grid in figures 4.1 and 4.3. `move_dir_i` and `move_dir_j` indicate the change in `i` and `j` needed to move the ant one step straight ahead. Consequently `move_dir_i` and `move_dir_j` also indicate the direction the ant is facing. To simplify visualization of the trail and an ant's progress along the trail, the common directions of north, east, south, and west are imposed on the trail. An ant's initial direction is considered to be easterly.

```

/* ----- */
/* Calculate fitness of an individual */
/* ----- */

int calc_fitness(int present_state)

/* argument must be start_state when function called */

/* fitness =
   the number of trail steps the ant can cover in 200 time steps
   Note: trail step is erased when an ant steps on it in order
   to prevent the retracing of trail steps.
```

Figure 4.14: Determine an Ant's Fitness (continued on next page)

```

FSM output = 0: turn 90 degrees to the right
              1: turn 90 degrees to the left
              2: move forward one step
              3: do nothing--permits change of state          */

/* The ordered pair (i, j) is the ant's location (row, column) in
   the trail array. The ordered pair (ii, jj) is the location of
   the step in front of the ant. The ordered pair (move_dir_i,
   move_dir_j) indicates how i and j change when the ant moves
   ahead 1 step. It also indicates the direction the ant is
   facing:

   when (move_dir_i, move_dir_j) =
       (0, 1)      ant is facing E
       (0, -1)     ant is facing W
       (1, 0)      ant is facing S
       (-1, 0)     ant is facing N          */
{
line #
1:      int i = 0, j = 0,
        /*ant starts in the upper left of the trail array */
2:      ii, jj,
3:      move_dir_i = 0, move_dir_j = 1,    /* ant starts facing E */
4:      fit = 0,                          /* fitness          */
5:      time = 0,
6:      future_state,
7:      input /* FSA input: pheromone content of trail step
              immediately ahead */;

8:      short int trail [32] [32] = {
          /* the John Muir trail */

9:      {0,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
10:     {0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
11:     {0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
12:     {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
13:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
14:     {1,1,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1},
15:     {0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
16:     {0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
17:     {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
18:     {0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
19:     {0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
20:     {0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0},
21:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0},
22:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0},
23:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0},
24:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0},
25:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0},
26:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0},
27:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0},
28:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0},
29:     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},

```

Figure 4.14 continued: Determine an Ant's Fitness (continued on next page)

```

30:  {0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
31:  {0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
32:  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
33:  {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
34:  {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
35:  {0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
36:  {0,0,0,0,0,1,0,0,0,1,1,1,1,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0}

37:  }; /* trail array */

38:  do { /*traverse trail if have not completed trail
        and time permits */
        /* ii and jj are the row and column of the step
        in front of the ant */
39:    ii = (i + move_dir_i) % 32; if (ii < 0) ii += 32;
40:    jj = (j + move_dir_j) % 32; if (jj < 0) jj += 32;

41:    input = trail [ii] [jj];
42:    future_state = next_state [present_state] [input];

43:    switch (output [present_state] [input]) {

44:        case 0: /* turn 90 degrees to the right */

                /* ant facing E or W */
45:        if (move_dir_i == 0) {
                /* change E to S, or W to N */
46:            move_dir_i = move_dir_j;
47:            move_dir_j = 0;
48:            break;
49:        }

                /* ant facing N or S */
50:        else {
                /* change N to E, or S to W */
51:            move_dir_j = -move_dir_i;
52:            move_dir_i = 0;
53:            break;
54:        }

55:        case 1: /* turn 90 degrees to left */

                /* ant facing E or W */
56:        if (move_dir_i == 0) {
                /* change E to N, or W to S */
57:            move_dir_i = -move_dir_j;
58:            move_dir_j = 0;
59:            break;
60:        }

```

Figure 4.14 continued: Determine an Ant's Fitness (continued on next page)

```

        /* ant facing N or S */
61:    else {
        /* change N to W, or S to E */
62:        move_dir_j = move_dir_i;
63:        move_dir_i = 0;
64:        break;
65:    }

66:    case 2: /* move 1 step ahead */ {

67:        i = ii;
68:        j = jj;
69:        if (trail [i][j]) {
        /* ant on marked trail step */
70:            ++fit; /* increase fitness */
71:            trail [i][j] = 0; /* erase trail step */
72:        }
73:    }
    /* case 3: do nothing--permits change of state */

74: } /* switch */

75: present_state = future_state;
76: ++time;

77: } while (time < 200 && fit < maxfitness);

78: if (fit == maxfitness) {
79:     fprintf(fp_perfect,
80:         "Generation %d; Start State=%d; ",
81:         generation, start_state);
82:     fprintf(fp_perfect,
83:         "Fitness of 89 reached in %d time steps.\n",
84:         time);
85:     save_FSM(start_state); /* save a perfect scorer */
86: }
87: ++fitness_count[fit]; /* fitness histogram */
88: return fit;

} /* calc_fitness*/

```

Figure 4.14 continued: Determine an Ant's Fitness

A **do-while** (lines 38-77) loop moves the ant along the trail. The ant's response to the input (marked step=1, unmarked step=0) (line 41) is controlled by a **switch** statement (line 43). A left or right turn requires changing the values of **move_dir_i** and **move_dir_j** (lines 44-65). To move onto the next step the ant's

location, indicated by *i* and *j*, is changed. The formulas (lines 39-40 and 67-68) reflect the top-bottom and left-right adjacency of the trail grid (array). When an ant moves onto a marked step (line 69), the ant's fitness is incremented (line 70) and the marking is removed from the trail step (line 71) to prevent the ant from backtracking on the trail.

An ant's fitness is the number of pheromone-marked trail steps which the ant traverses. Since the trail has 89 marked trail steps, the optimal fitness is 89. An ant is moved along the trail until it attains a fitness of 89, or until it has spent 200 time steps following the trail. Hence this is the criteria for terminating the loop (line 77). An ant has achieved a perfect score if it attains a fitness of 89; the FSM representing a perfect scoring ant models a successful solution to the problem. Perfect scoring FSMs are output to the file with the **ps** extension (lines 78-86). The ant's fitness is reflected in the fitness histogram (line 87) before returning the fitness to the calling program (line 88). The **calc_fitness** procedure can be easily rewritten for other types of problems.

calc_fitness calls **save_FSM** (figure 4.15) to record the perfect scoring FSMs in a *.ps file. Only those states which can be reached via a transition are printed out. For example, for the FSM in table 4.1, **save_FSM** will print out only the four states which appear in the table. The **reachable** array indicates whether a state is reachable via some sequence of transitions from the **start_state**. **reachable[i] = 1** if state *i* is reachable (lines 4, 9 and 14); otherwise it is 0 (line 1). As each state is reached for the first time it is placed on the **queue** (lines 3, 7-8, and 12-13) so that its next states can be tagged as reachable. The reachable states

are output to the *.ps file (lines 18-29).

```

/* ----- */
/*      Save Finite State Machine      */
/* ----- */

void save_FSM(int start_state)
/* print out state transitions and ouput in table form only
   (5 per row) for those states which are reachable from the
   start state */
{
line #
1:   int i,j, reachable[max_num_states]={0},
2:       first=0, last=0, queue[max_num_states];

3:   queue[0] = start_state;
4:   reachable[start_state]=1;

   /* breadth first search approach
      to finding which states are reachable */

5:   while (first <= last) {

6:       if (!reachable [next_state [queue[first]][0]]) {
7:           ++last;
8:           queue [last] = next_state [queue[first]][0];
9:           reachable [queue[last]] =1;
10:      }

11:      if (!reachable [next_state[queue[first]][1]]) {
12:          ++last;
13:          queue [last] = next_state [queue[first]][1];
14:          reachable [queue[last]] =1;
15:      }

16:      ++first;

17:  } /* while */

   /* print out state transition table for those states reachable
      from the start state, 5 in a row */

18:   for (i=0; i<5; ++i) fprintf(fp_perfect, "ps fs/output  ");
19:   fprintf(fp_perfect, "\n");

20:   for (i=0; i<5; ++i) fprintf(fp_perfect, "    in=0:in=1  ");

21:   for (i=0, j=0; i<max_num_states; ++i)

```

Figure 4.15: Output Successful FSMs (continued on next page)

```

22:         if (reachable[i]) {
23:             if (j==0) fprintf(fp_perfect, "\n");
24:             fprintf(fp_perfect, "%2d%3d/%1d:%2d/%-4d",
25:                 i, next_state[i][0], output[i][0],
26:                 next_state[i][1], output[i][1]);
27:             j = ++j % 5;
28:         }
29:         fprintf(fp_perfect, "\n");

    } /* save_FSM */

```

Figure 4.15 continued: Output Successful FSMs

```

/*****
/*
/*          This is the basic driver for the GA          */
/*          The initial data validation tests are FSM specific      */
/*
*****/

void main(void)
{
line #
1:     int i, j;

2:     if (!validate_data()) return;
3:     print_header(&fp_stats, &fp_perfect);

    /* the actual GA starts here */
4:     initialize(); /* generation 0 */
5:     save_stats();

6:     for (generation=1;
        /* maximum fitness has not been reached
        and more generations available */
7:         (generation <= last_generation) &&
8:         (fitness [sorted[row_of_sorted][0]] < maxfitness);
9:         ++generation) {
10:        create_next_generation();
11:        save_stats();

12:    }

13:    prologue();

    } /* main */

```

Figure 4.16: The main Part of the GA

The **main** procedure (figure 4.16) of the GA consists of four parts. First **validate_data** (figure 4.17, invoked on line 2 of figure 4.16) checks to make sure that the values assigned to the constants do not contradict each other. For example,

```
#define bits_per_genome 50
#define bytes_per_genome 6
```

would cause an error message (lines 22-29 of figure 4.17, invoked on line 2 of figure 4.16) because seven bytes are needed to store a 50-bit genome. Then **print_header** (figure 4.18, invoked on line 3 of figure 4.16) writes the headings to the two output files. **prologue** (figure 4.19, invoked on line 13 of figure 4.16) closes out those files when the program is finished executing. **main** (figure 4.16) starts the GA and carries out the iterative evolutionary process.

```

/* ----- */
/*      Validate Value of Constants      */
/* ----- */

int validate_data(void)
/* tests to validate values assigned to constants
   as not contradictory      */
{
line # 1:      int calculated;

        /* TEST #1: Are there enough bits for storing the number
           of each state?      */

2:      calculated = power_of_2(bits_per_state_num);
           /* number of bits necessary */
3:      if (max_num_states != calculated) {
4:          printf(
5:              "(%d) max_num_states not=(%d) 2^bits_per_state_num \n",
6:              "make corrections and try again\n",
7:              max_num_states,calculated);
8:          return 0;
9:      }

```

Figure 4.17: Check for Contradictory Constant Values (continued on next page)

```

/* TEST #2: Are there enough bits for storing the
   entire genome? */

10:   calculated = /* number of bits necessary */
11:         bits_per_state_num + num_of_inputs * max_num_states
12:         * (bits_per_state_num + bits_per_output);
13:   if (bits_per_genome != calculated) {
14:       printf("(%d) bits_per_genome not =\n",
15:             4*(%d) bits_per_state_num +\n",
16:             "      num_of_inputs * max_num_state\n",
17:             "      * (bits_per_state_num + bits_per_output)\n",
18:             "make corrections and try again\n",
19:             bits_per_genome, calculated);
20:       return 0;
21:   }

/* TEST #3: Are there enough bytes for storing the
   entire genome? */

22:   calculated=(bits_per_genome+7)/8; /* number of bits necessary */
23:   if (bytes_per_genome != calculated) {
24:       printf(
25:           "(%d) bytes_per_genome not=(%d) (bits_per_genome + 7)/8\n",
26:           "make corrections and try again\n",
27:           bytes_per_genome, calculated);
28:       return 0;
29:   }

30:   return 1;
} /* validate_data */

```

Figure 4.17 continued: Check for Contradictory Constant Values

```

/* ----- */
/*      Print Header on Output Files      */
/* ----- */

void print_header(void)
/* print out header information for the GA in
   the statistics and perfect score files */
{
    fp_stats = fopen(printed_output, "w");
    fp_perfect = fopen(perfect_score, "w");

    fprintf(fp_stats, "%s\nStatistics for run #%d on file %s\n",
            run_name, run_num, printed_output);
    fprintf(fp_perfect,
            "%s\nPerfect scorers for run #%d on file %s\n",
            run_name, run_num, perfect_score);
}

```

Figure 4.18: Prints Headings on Output Files (continued on next page)

```

fprintf(fp_stats, "Seed=%.8f\n", seed);
fprintf(fp_perfect, "Seed=%.8f\n", seed);
fprintf(fp_stats,
        "Crossover rate=%.2f;      Mutation rate=%.2f\n",
        xover_rate, mutation_rate);
fprintf(fp_perfect,
        "Crossover rate=%.2f;      Mutation rate=%.2f\n",
        xover_rate, mutation_rate);
fprintf(fp_stats, "Population size=%d\n", popsize);
fprintf(fp_perfect, "Population size=%d\n", popsize);
fprintf(fp_stats, "Select the top %d individuals or %.2f%%",
        select, 100. * (double)select / (double)popsize);
fprintf(fp_perfect, "Select the top %d individuals or %.2f%%",
        select, 100. * (double)select / (double)popsize);
fprintf(fp_stats, " of each generation for reproduction.\n\n");
fprintf(fp_perfect, " of each generation for reproduction.\n\n");
fprintf(fp_stats, " Gen    Max/# with Max    ");
fprintf(fp_stats,
        "Average/Median/Mode/# with Mode    Standard dev.\n");

} /* print header information */

```

Figure 4.18 continued: Prints Headings on Output Files

```

/* ----- */
/*          Close Output Files          */
/* ----- */

void prologue(void)
/* close the statistics and perfect score files */
{
    fprintf(fp_stats, "\n");
    fprintf(fp_perfect, "\n");

    fclose(fp_stats);
    fclose(fp_perfect);
} /* prologue */

```

Figure 4.19: Close Output Files

Typically the `initialize` procedure (figure 4.20, invoked on line 4 of figure 4.16) starts the random number generator (line 2, figure 4.20) and uses it to create the initial population. Generation 0 is created (lines 4-5, figure 4.20) by generating `bytes_per_genome=57` bytes (with values of 0 through 255) as opposed to

bits_per_genome=453 bits. The extra bits which result are discarded when each child is bred for generation 1. The **initialize** procedure also prepares generation 0 to parent generation 1. Invoking procedure **bytestring_to_FSM** (line 6, figure 4.20) converts a single bytestring to its corresponding FSM table. The table is used by **calc_fitness** (line 7, figure 4.20) to determine the fitness for a single individual. The **sorted** array is initialized to prepare for the sorting of the population (line 8). After the population has been processed in this way, the population is sorted in decreasing order of fitness (line 10). To avoid moving the large genomes around in memory, the indices, which indicate the location of each individual in the **pop** array, are sorted. After the population is sorted, it is ready to parent the next generation.

```

/* ----- */
/*           Initialize population           */
/* ----- */

void initialize(void)
{
line #
1:   int i, j;

      /* Randomly generate initial population */
2:   Randomize(); /* initialize random number generator */

3:   for (i=0; i < popsize; ++i) {
4:       for (j=0; j < bytes_per_genome; ++j)
5:           pop [new_pop] [i] [j] = Rnd(0, 255);
6:       bytestring_to_FSM(&start_state, &pop[new_pop][i][0]);
7:       fitness[i] = calc_fitness(start_state);
8:       sorted[0][i]=i;
9:   }

10:  row_of_sorted=sort(0, popsize-1, 0); /*by decreasing fitness*/

    } /* initialize */

```

Figure 4.20: Initialization

Control is then returned to **main** (figure 4.16) which then calls **save_stats** (figure 4.21, invoked on line 11 of figure 4.16). **save_stats** computes and outputs the statistics for a generation. First the **maximum** fitness is set (line 3, figure 4.21) and the number of individuals with that fitness is determined (lines 4-6). The **median** is located at the center of the sorted array (lines 7-12). While **popsize** for this program is an even number (lines 8-10), the coding allows for it to be odd (lines 11-12). The calculations for the average (lines 13-15) and standard deviation (lines 16-18) are straight forward. The fitnesses are added together and divided by **popsize** to calculate the average (lines 13-15). The standard deviation is similarly coded utilizing the built in functions of **pow** and **sqrt** (lines 16-18). **fitness_count[i]** is a count of the number of FSMs in the population which have a fitness of **i**. This is used to find the **mode** (lines 19-26). The code to find the **mode** allows for a multimodal possibility. There will be at least a single **mode** (lines 20-23). **multimode** counts the number of additional **modes** (lines 24-25). After the statistics for the **generation** are recorded on the ***.st** file (lines 27-29), **multimode** permits the recording of all the modes (lines 30-37). If **multimode=0**, there was only one **mode** and the single line which was output (lines 27-29) fully describes the statistics for the generation.

```

/* ----- */
/*           Save Statistics           */
/* ----- */

void save_stats(void)

```

Figure 4.21: Calculating and Outputting the Statistics for Each Generation (continued on next page)

```

line # {
1:      int max, num_of_max=1, mode=maxfitness, multimode=-1, i, j, k;
2:      double avg=0, median, standard_dev=0;

           /* ----- */
           /*      Determine maximum      */
           /* ----- */

3:      max = fitness[ sorted[row_of_sorted][0] ];
4:      while (num_of_max<popsize
5:             && max==fitness[ sorted[row_of_sorted][num_of_max] ])
6:          ++num_of_max;

           /* ----- */
           /*      Determine median      */
           /* ----- */

7:      i = popsize/2;
8:      if (2*i == popsize)
9:          median = (double) (fitness[sorted[row_of_sorted][i-1]] +
10:                             fitness[sorted[row_of_sorted][i]])/2.0;
11:      else
12:          median = (double) fitness[ sorted[row_of_sorted][i] ];

           /* ----- */
           /*      Determine average      */
           /* ----- */

13:     for (i=0; i<popsize; ++i)
14:         avg += (double) fitness[i];
15:     avg /= (double) popsize;

           /* ----- */
           /*      Determine standard dev.  */
           /* ----- */

16:     for (i=0; i<popsize; ++i)
17:         standard_dev += pow( (double) fitness[i] - avg, 2.);
18:     standard_dev = sqrt(standard_dev/ (double) popsize);

           /* ----- */
           /*      Determine mode      */
           /* ----- */

19:     for (i=maxfitness, j=5; i>=0; --i) {
20:         if (fitness_count[mode] < fitness_count[i]) {
21:             mode=i;
22:             multimode=0;
23:         }
24:         else
25:             if (fitness_count[mode]==fitness_count[i]) ++multimode;
26:     }

```

Figure 4.21 continued: Calculating and Outputting the Statistics for Each Generation (continued on next page)


```

/* ----- */
/*           Print Statistics           */
/* ----- */

27:     fprintf(fp_stats, "%4d%6d/%-10d%10.4f/%5.1f /%3d /%-11d%11.4f\n",
28:             generation, max, num_of_max, avg, median,
29:             mode, fitness_count[mode], standard_dev);
30:     if (multimode)
31:         for (j=0, i=mode; j<multimode; ++j) {
32:             --i;
33:             while(i >= 0 && fitness_count[mode]!=fitness_count[i]) --i;
34:             if (i >= 0)
35:                 fprintf(fp_stats, "%31.4f/%5.1f /%3d /%-11d\n",
36:                         avg, median, i, fitness_count[i]);
37:         }

    } /* save_stats */

```

Figure 4.21 continued: Calculating and Outputting the Statistics for Each Generation

```

/* ----- */
/*           Create next generation     */
/* ----- */

void create_next_generation(void)
{
line #
1:     int fit, i;

2:     for (i=0; i<=maxfitness; ++i) fitness_count[i]=0;
3:     old_pop = new_pop; new_pop = ++new_pop % 2;

4:     for (i=0; i<popsizes; ++i) {
5:         xover(&pop[new_pop][i][0]);
6:         mutate(&pop[new_pop][i][0]);
7:         bytestring_to_FSM(&start_state, &pop[new_pop][i][0]);
8:         fitness[i] = calc_fitness(start_state);
9:     }

10:    for (i=0; i<popsizes; ++i) sorted[0][i] = i;
11:    row_of_sorted = sort(0, popsizes - 1, 0); /*by decreasing fitness*/

    } /* create_next_generation */

```

Figure 4.22: Breeding the Next Generation

Each time through the **for** loop of **main** (lines 6-12, figure 4.16) yields a new **generation**. The iterative process halts when at least one perfect scoring ant has evolved (line 8) or when **last_generation** generations have been bred (line 7). For some experiments (described in chapter 8.1.1) the test on line 8 is eliminated and the GA breeds **last_generation** generations, even when a perfect scoring FSM is located in an earlier generation. **create_next_generation** (figure 4.22, called on line 10 of figure 4.16) initializes the fitness histogram (line 2, figure 4.22), and exchanges the values of **old_pop** and **new_pop** (line 3, figure 4.22) to label the previous generation as the parents and indicate where the newborns are to be placed.

The **xover** procedure (figure 4.23, invoked on line 5 of figure 4.22) selects a mating pair from the top **select** members of the population (lines 3-4, figure 4.23) to produce a single child. One parent is the bit donor indicated by **present_parent** (line 1). Each time a bit is copied to the child (line 10), a random number between 0 and 1 is generated (line 8). Whenever this number falls below **xover_rate** (line 8), a crossover point is created by changing the donating parent (line 9). The **mask** permits a single bit to be copied from the **present_parent** to the **child** (lines 6 and 10-11). As in **unpack_genome** (figure 4.9), the **mask** contains a single 1 (line 6, figure 4.23) which is shifted through the eight bits of the byte (line 7, figure 4.23). ANDing **mask** with a byte from the **parent** isolates the desired bit (line 10-11, figure 4.23). ORing this with the corresponding byte of the **child** (lines 10-11), which was initialized to zeroes (line 6), copies the bit from the **parent** to the **child**. **i** is the index into the bytestrings of the **parent** and **child**, **j** is the bit count, and **k** counts the bits in the **i**th byte

(lines 5-12). The `for` loops (lines 5-12) move along the genome processing eight bits per byte for 453 bits.

```

/* ----- */
/*               Crossover               */
/* ----- */

void xover(packed_genome child)
/* selects a pair of parents and mates them to produce a single
   child */
{
line #
1:   int i, j, k, parent[2], present_parent=0;
2:   byte mask;

   /* i = byte position in genome
      j = bit position in genome */

3:   k = Rnd(0, select - 1); parent[0] = sorted[row_of_sorted][k];
4:   k = Rnd(0, select - 1); parent[1] = sorted[row_of_sorted][k];

5:   for (i=0, j=0; j < bits_per_genome; ++i)
6:       for (k=0, mask=0x80, child[i]=0;
7:           k<8 && j<bits_per_genome; ++k, ++j, mask >>= 1) {
8:           if (Random() <= xover_rate) /* change present parent */
9:               present_parent = (++present_parent) % 2;
10:          child[i] |=
11:              mask & pop[old_pop][parent[present_parent]][i];
12:      }
} /* xover */

```

Figure 4.23: Mating

```

/* ----- */
/*               Mutate               */
/* ----- */

void mutate(packed_genome bytestring)

/* ^ is EXOR operator:
   0^x=x , i.e. 0 preserves value; 0 is the identity element.
   1^x=x', i.e. 1 inverts value.

```

Figure 4.24: Mutation (continued on next page)

```

    Therefore, mask^byte inverts the single position in byte
    where mask has a 1.
    */
{
line #
1:     int i, j, k;
2:     byte mask;

    /* i = byte position in genome
       j = bit position in genome */

3:     for (i=0, j=0; j < bits_per_genome; ++i)
4:         for(k=0,mask=0x80;k<8 && j<bits_per_genome;++k,++j,mask>>= 1)
5:             if (Random() <= mutation_rate)
6:                 bytestring[i] = mask ^ bytestring[i];

    } /* mutate */

```

Figure 4.24 continued: Mutation

Mutation (figure 4.24) is then applied (on line 6 of figure 4.22, `create_next_generation`) at `mutation_rate` per bit. In the mutation procedure `i`, `j`, and `k`, and their loops (lines 3-4) serve the same purposes as they do in `xover` (lines 5-7, figure 4.23). In `mutate` (figure 4.24), a random number is generated for each bit (line 5). To implement the `#defined mutation_rate`, a bit is inverted when the corresponding random number falls below `mutation_rate` (lines 5-6). The EXOR can either retain a value, or invert (NOT) it. Zero is the identity element for EXOR, therefore when one operand of an EXOR is a zero the other operand is returned as the answer. When one operand of an EXOR is a 1, the value of the other operand is inverted. Here again the `mask` contains a single 1 (line 4) which is used to invert a bit (line 6). When a bit is to be mutated, the `mask` is EXORed with the present byte. The 1 in the `mask` inverts the indicated bit, and the zeroes in the remainder of the `mask` retain the seven other bits. The rest of `create_next_generation` (lines 7-11, figure 4.22) prepares these children to

parent the next generation. The process is identical to the one carried out by **initialize** (figure 4.20), but the initialization of the **sorted** array is moved outside of the main loop of **create_next_generation** (line 10, figure 4.22) in order to retain the results of the sort of the parents while the next generation is being bred. When control is returned to **main**, **save_stats** (figure 4.21, called on line 11 of figure 4.16) calculates the statistics for the current generation and outputs it to the file with the **st** extension.

Figure 4.25 contains a listing of an **st** file, and figure 4.26 is the corresponding **ps** file. All numbers in figure 4.25, except for those in the first column, are related to fitness. The column labelled **Max/# with Max** is the maximum fitness appearing in the generation and the number of individuals in the generation which have that fitness. Similarly for **Mode** and **# with Mode**.

```
Base case modelled after Jefferson et al.
Statistics for run #3 on file r3b.st
Seed=0.69343845
Crossover rate=0.01;      Mutation rate=0.01
Population size=65536
Select the top 3277 individuals or 5.00% of each generation for
reproduction.
```

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
0	64/1	3.7148/ 1.0 / 0 /26956	7.2664
1	79/1	7.8863/ 2.0 / 0 /18636	11.6169
2	67/1	11.7417/ 3.0 / 0 /16454	15.2043
3	65/1	14.1120/ 5.0 / 0 /15110	16.6865
4	75/1	15.9189/ 10.0 / 42 /15638	17.4709
5	79/1	17.4440/ 10.0 / 42 /18097	18.0109
6	79/1	18.4360/ 10.0 / 42 /18899	18.1688
7	80/1	19.0267/ 10.0 / 42 /18411	18.2153
8	81/1	18.5346/ 10.0 / 42 /13883	17.9810
9	81/1	18.3342/ 10.0 / 0 /10563	18.0576
10	81/1	19.8597/ 11.0 / 0 /9674	18.6749
11	81/1	21.6483/ 14.0 / 0 /8610	19.2626

Figure 4.25: Listing of File with **st** Extension from Sample Run (continued on next page)

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
12	81/2	23.7276/ 21.0 / 42 /8057	20.0531
13	81/5	25.1715/ 22.0 / 42 /8555	20.5392
14	81/12	27.8097/ 32.0 / 42 /8688	21.2958
15	82/1	31.9812/ 35.0 / 42 /9499	22.5329
16	83/1	42.3225/ 42.0 / 74 /17242	26.0879
17	83/1	41.4549/ 42.0 / 74 /15167	26.9132
18	83/4	37.0724/ 35.0 / 42 /8977	26.4808
19	83/12	38.0038/ 37.0 / 80 /11992	27.4448
20	83/27	40.4289/ 42.0 / 80 /10893	28.2652
21	83/48	43.4303/ 42.0 / 81 /14017	29.0688
22	83/93	43.6623/ 42.0 / 81 /14858	29.2561
23	84/3	43.3132/ 42.0 / 81 /13922	29.2306
24	84/3	44.5001/ 42.0 / 81 /9594	29.1818
25	84/20	44.8606/ 42.0 / 82 /9478	29.2679
26	84/58	50.9981/ 47.0 / 83 /23495	29.8806
27	84/146	55.1226/ 58.0 / 83 /27333	29.0099
28	85/2	56.2042/ 58.0 / 83 /28122	28.7117
29	85/5	54.1236/ 52.0 / 83 /25421	29.1874
30	86/2	48.1696/ 42.0 / 83 /16582	29.7283
31	86/5	54.6107/ 52.0 / 84 /23666	29.6827
32	86/6	57.0353/ 79.0 / 84 /26154	29.2422
33	86/10	57.2668/ 79.0 / 84 /25964	29.2187
34	86/29	55.3385/ 57.0 / 84 /22706	29.5564
35	86/78	50.7199/ 43.0 / 84 /12132	30.0218
36	86/217	51.5211/ 44.0 / 85 /16674	29.8535
37	87/2	52.2516/ 46.0 / 85 /17704	29.6856
38	88/1	52.0490/ 46.0 / 85 /16301	29.6341
39	88/2	50.4423/ 43.0 / 85 /8220	29.2398
40	88/3	53.4057/ 47.0 / 86 /18050	28.7444
41	88/13	54.2980/ 52.0 / 86 /18943	28.7117
42	88/37	54.8068/ 52.0 / 86 /19261	28.6745
43	88/136	54.8835/ 52.0 / 86 /18390	28.5455
44	88/477	53.7232/ 50.0 / 86 /15006	28.4901
45	89/1	49.9201/ 43.0 / 87 /8672	28.0389

Figure 4.25 continued: Listing of File with **st** Extension from Sample Run

Base case modelled after Jefferson et al.
 Perfect scorers for run #3 on file r3b.ps
 Seed=0.69343845
 Crossover rate=0.01; Mutation rate=0.01
 Population size=65536
 Select the top 3277 individuals or 5.00% of each generation for reproduction.

Generation 45; Start State=4; Fitness of 89 reached in 197 time steps.

ps fs/output	ps fs/output	ps fs/output	ps fs/output	ps fs/output
in=0:in=1	in=0:in=1	in=0:in=1	in=0:in=1	in=0:in=1
0 22/2: 4/2	1 10/0: 9/2	2 8/3: 1/2	4 10/0: 4/2	6 1/2: 6/2
7 6/0: 8/2	8 6/2: 6/2	9 7/0:22/2	10 9/0: 0/2	22 25/2:22/2
25 7/1: 2/0				

Figure 4.26: Listing of File with **ps** Extension from Sample Run

With the benchmark program complete, additional strategies can be added. The next chapter describes two new algorithms, and other modifications to the benchmark. These two new strategies are mutually exclusive. Each of these algorithms reorganizes the genomes in the parent pool during run time in order to enhance schema development.

>>>●<<<

Chapter 5

The new operators

Two new reorganizing operators were designed for this thesis. Both try to "line up" states in order to create schemata out of features, and to enhance the GA's efficiency. They are FSM specific and not biologically based. From the biological perspective, genes which code for a given characteristic have fixed positions in the genome, but the nature of an FSM defies the natural phenomenon. For example the next states and output for the start state can be located in any of 32 locations within the genome, as can the same information for its next states. The new operators try to compensate for this deficiency in the FSM.

The first reorganization operator, called SFS, sandardizes the location of the next/future state for some of the present state/input pairs. The second reorganization operator, called MTF, compresses the relevant data within the genome by systematically moving it to the front of the genome. Each operator standardizes equivalent FSMs with the same number of reachable states. Since no minimization is occurring with these two operators, a whole equivalence class is not reduced to a single representative of the class, only equivalent FSMs with the same number of states are altered to the same representation. Both algorithms standardize the start state to q_0 . In both cases, since the start state is always q_0 , with the Jefferson (figure 4.6) genome

innovation due to crossover is lost with respect to the start state. Even though the start state will always be q_0 after reorganization, the first five bits of the genome will still contain the start state. There are two reasons for doing this:

- 1) the initialization process permits any state to be the start state, and
- 2) mutation will still be able to change the start state.

Also at some time in the future, testing could apply the operators to each member of the parent pool with some probability as opposed to every member of the parent pool. With only some members of the parent pool reorganized, q_0 would not necessarily be the start state for each genome. The next two sections describe the new algorithms and their coding.

5.1 The SFS algorithm

The SFS algorithm reorganizes a finite state machine by standardizing the transitions to the future/next states according to a mathematical function. The purpose of standardizing transitions is to enhance schema growth. In the next two sections the SFS algorithm and its coding are described, and two SFS reorganizations are presented.

5.1.1 SFS: description and coding

In order to standardize transitions, SFS has to rename the states. Therefore it has to move the next state and output for a renamed state to the proper location in the genome based on its new state name. It also has to change all references to

```

/* ----- */
/*           Switch Rows           */
/* ----- */

void switch_rows(int row1, int row2, table tbl)
{
line #
1:     int i, temp;

2:     for (i=0; i<num_of_inputs; ++i) {
3:         temp = tbl[row1][i];
4:         tbl[row1][i] = tbl[row2][i];
5:         tbl[row2][i] = temp;
6:     }

    } /* switch rows */

```

Figure 5.1: Interchange Rows of a Table

```

/* ----- */
/*           Exchange States         */
/* ----- */

void xchange_states(int statel, int state2)
{
line #
1:     int i,j;

2:     if (statel != state2) {

3:         switch_rows(statel, state2, next_state);
4:         switch_rows(statel, state2, output);

5:         for (i=0; i<max_num_states; ++i)
6:             for (j=0; j<num_of_inputs; ++j)
7:                 if (next_state[i][j] == statel)
8:                     next_state[i][j] = state2;
9:                 else
10:                    if (next_state[i][j] == state2)
11:                        next_state[i][j] = statel;

12:    }

    } /* xchange_states */

```

Figure 5.2: Interchange States Based on Name Change

the renamed state in the next state part of the FSM table. Since there is limited space in the genome, a state, which is assigned a new name, has its name and location exchanged with the state which has the desired name. For example, if SFS reassigns state q_{20} to state q_4 , the 20th and 4th rows of the FSM table must be interchanged, and their names in the next state part of the table must also be exchanged. Procedures **xchange_states** and **switch_rows** perform all necessary exchanges. **switch_rows** (figure 5.1) exchanges the data in two rows of a table. It is called by **xchange_states** (figure 5.2) which not only exchanges the rows of data (lines 3-4, figure 5.2) but also appropriately changes the next state names in the **next_state** table (lines 5-11, figure 5.2). The **for** loops (lines 5-6) move through the states, and through the inputs for each state.

```

/* ----- */
/*           Reorganization           */
/* ----- */

void reorganizeFSM(void)
{
line #
1:   int i;

2:   for (i=0; i<select; ++i) {
3:       bytestring_to_FSM (&start_state,
4:                           &pop[new_pop][sorted[row_of_sorted][i]][0]);
5:       SFS (&start_state);
6:       FSM_to_bytestring (start_state,
7:                           &pop[new_pop][sorted[row_of_sorted][i]][0]);
8:   }

} /* reorganizeFSM */

```

Figure 5.3: Reorganize Parent Pool

The **SFS** procedure, called by **reorganizeFSM** (figure 5.3), utilizes the

procedures in figures 5.1 and 5.2 to reorganize the FSM. **reorganizeFSM** loops through the top **select** individuals (line 2) for their reorganization. There is no need to reorganize those FSMs which cannot be selected as a parent. Since **SFS** operates on a table, the genome must be converted to a table (lines 3-4) before **SFS** is invoked (line 5), and then returned to the bytestring form for storage (lines 6-7).

SFS was designed with the intent of enhancing schema growth by standardizing transitions to the next state. A mathematical function determines the desired next state for each present state/input pair. A finite state machine with 32 states and two inputs will have 64 transitions to next states. Since the genome has only 32 states, each state becomes the desired next state for two present state/input pairs. In addition the start state is standardized to q_0 . Consequently **SFS** consists of three steps. First the initial state is exchanged with state q_0 , and q_0 is placed on a queue so its future states can be reassigned.

The second step (figure 5.5) of **SFS** (figure 5.6) processes the state at the front of the queue **to_be_processed2** (lines 3-4, figure 5.5) by trying to assign its next state names based on a mathematical function. Most of the state reassignments, sometimes all of them, are carried out by **step2**. This is the most complex part of the SFS algorithm. This algorithm was designed for input $j \in \{0,1\}$ and is not applicable for other values of j . Next states are assigned by the following multi-part mathematical function. The argument $\text{cut\_off} = \frac{\text{max\_num\_states}}{2}$ (line 12, figure 5.6). Then state i with input j has desired next state =

- 1) $2i+j+1$ when $i \leq \text{cut\_off}-2$ (line 7, figure 5.5),
- 2) $\text{max\_num\_states}-1$ when $i = \text{cut\_off}-1$ (lines 8-9, figure 5.5), and
- 3) $2(\text{max\_num\_states}-2-i)+j+1$ for

$\text{cut\_off}-1 < i < \text{max\_num\_states}-1$ (line 11, figure 5.5).

Note that state $\text{max\_num\_states}-1$ is not included in this function, and for state $\text{cut\_off}-1$ the desired next state is the same for the two inputs. As implemented in the program, the desired next state for state $\text{max\_num\_states}-1$ is set to 0 (line 10). This case will be covered shortly.

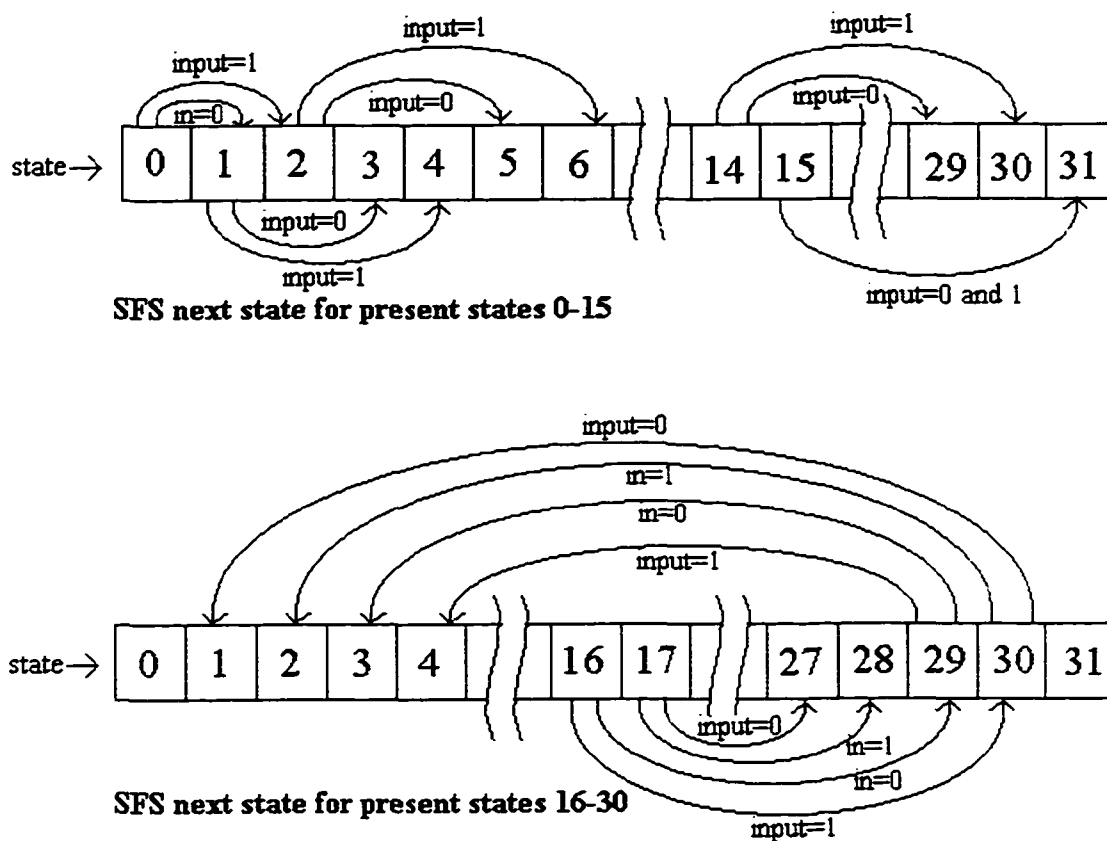


Figure 5.4: SFS Function Assigning Next State for a Given Present State/Input

The SFS function which assigns next states is represented graphically in figure 5.4 for a 32-state genome. As indicated, the next states for state 0 are states 1 and 2. For state $i \in \{1, 2, 3, \dots, 14\}$, the assigned next states are $n+1$ and $n+2$, where

n is the assigned next state for state $i-1$ with input 1. Conflict in next state assignments can occur for states 15 through 30. For state 15 the assigned next state is state 31 for both input values. The assigned next states for state 16 are states 29 and 30—the same next states the function assigns for state 14. For state $i \in \{17, 18, 19, \dots, 30\}$, the assigned next states are $n-1$ and $n-2$, where n is the assigned next state for state $i-1$ with input 0. As indicated earlier in the function description, state 31 has no assigned next states. The next state assignment for state 31 and for conflicts, when they occur, are resolved by the third step of SFS. This will be described shortly.

```

/* ----- */
/*           Step 2 of SFS           */
/* ----- */

void step2 (int cut_off, int *next_step2,
            int *last_step2, int *last_step3,
            int used[], int to_be_processed2[],
            int to_be_processed3 [2*max_num_states] [2])
{
line #   int i, j, new_next_state;
1:
2:   while (*next_step2 <= *last_step2) {
3:       i = to_be_processed2[*next_step2];
4:       ++*next_step2;
5:       for (j = 0; j < num_of_inputs; ++j)
6:           if (!used[next_state[i][j] + 1]) {
7:               /* next state has not been reassigned */
8:               if (i < cut_off-1) new_next_state = 2*i + j + 1;
9:               else if (i == cut_off-1)
10:                  new_next_state = max_num_states-1;
11:               else if (i == max_num_states-1) new_next_state = 0;
               /* since state 0 is always used, the last state
                will always be processed by step 3 */
               else new_next_state = 2*(max_num_states-2-i) + j + 1;

```

Figure 5.5: The Second Step of the SFS Algorithm (continued on next page)

```

12:         if (used[new_next_state + 1]) {
13:             /* save state for processing by step 3 */
14:             ++*last_step3;
15:             to_be_processed3[*last_step3][0] = i;
16:             to_be_processed3[*last_step3][1] = j;
17:         }
18:         else {
19:             xchange_states(next_state[i][j], new_next_state);
20:             used[new_next_state+1] = 1;
21:             ++*last_step2;
22:             to_be_processed2[*last_step2] = new_next_state;
23:         }
24:     } /* while */
} /* step2 */

```

Figure 5.5 continued: The Second Step of the SFS Algorithm

The **while** loop (lines 2-24) in **step2** (figure 5.5) processes the states which have been put on the queue **to_be_processed2**. This queue contains the states which have been reassigned new state names (numbers) and which have to be processed by **step2** for next state reassignments. **next_step2** and **last_step2** (line 2) are respectively the positions of the first and last elements on the queue. The queue, implemented with the **to_be_processed2** array, is sufficiently large so that a cyclic queue will not be needed; that is, **to_be_processed2** will never have to store an element after the last position in the array is filled.

The **for** loop of **step2** on line 5 makes sure next states are reassigned for both inputs. When state **n** is reassigned, it is tagged by setting **used[n+1] = 1** (line 19) to indicate that reassignment has occurred. This prevents a state from again being reassigned if it is the next state for more than one present state-input pair. It also prevents a second present state-input pair from mapping into an already assigned

position since the reassignment function is two-to-one.

Looking at figure 5.5, if the next state for a given present state-input pair has been reassigned there is nothing to do. The test for the need to reassign the next state is on line 6. When reassignment is required an **if/else if/else if/else** (lines 7-11) determines the desired **new_next_state** based on the mathematical function. Then an **if** (line 12) determines whether **new_next_state** is available. When **new_next_state** is available (lines 17-22), it is exchanged with the **next_state** (line 18). **new_next_state** is then tagged as **used** (line 19), and added to the queue for processing by **step2** (lines 20-21). On the other hand, when **new_next_state** has already been **used** (lines 12-16), present state **i** with input **j** are stored on queue **to_be_processed3** for next state reassignment by step 3 of **SFS**.

step2 is a separate subroutine because it is invoked from two different places of **SFS** (lines 14 and 33, figure 5.6). The **SFS** algorithm (figure 5.6) consists of three steps. The first step standardizes the **start_state** as state 0. **start_state** and state 0 are interchanged (lines 6 and 8). State 0 is then tagged as reassigned (line 7) and placed on the queue for processing by the second step (lines 9-10). The queue for the third step of **SFS** is initialized (line 11). Then **step2** is called (line 13). It executes till its queue is empty (line 2, figure 5.5). Then step 3 (lines 15-35, figure 5.6) processes the present state-input pairs on its queue. **next_step3** and **last_step3** (line 15, figure 5.6) are respectively the positions of the first and last elements on the queue for step 3. The queue, implemented with the array **to_be_processed3**, is sufficiently large so that a cyclic queue will not be needed;

that is, `to_be_processed3` will never have to store an element after the last position in the array is filled. Between the time a present state-input pair is placed on `to_be_processed3` and the time it is taken off the queue for processing by step 3 (lines 16-18), the `next_state` could have been reassigned. Consequently, the `next_state` must be checked to see if it has been reassigned (line 19). When it has not been, step 3 looks for the closest available state in the genome (lines 20-27). This will be the `new_next_state.k` (lines 20-21) indicates the first available state to the left of the present state, and `n` (lines 22-23) indicates the closest available state to its right. Whichever is closer to the present state is the `new_next_state` (lines 24-27). `next_state` and `new_next_state` are interchanged (line 28). `new_next_state` is then tagged as `used` (line 29), and added to the queue for processing by `step2` (lines 30-31). `SFS` requires that reassignment of next states is always first attempted by `step2`; a state is reassigned by step 3 only when `step2` has failed. Therefore `step2` is called (line 32) to process this newly assigned state and any states which are reassigned during this execution of `step2`. Step 3 continues processing the states on its queue when `step2`'s queue is again empty.

```

/* ----- */
/*   Standardize Next (Future) States   */
/* ----- */

void SFS (int *start_state)
/* step 1: Make state 0 the start state.
  step 2: Standardize some present state/next state pairs on the
           genome.
           Let cut_off=max_num_states/2.
           Then, for state i with input j = 0,1, next state k is

```

Figure 5.6: The SFS Algorithm (continued on next page)

```

        k = 2i+j+1                                for i <= cut_off-2
        k = max_num_states-1                      for i = cut_off-1
        k = 2(max_num_states-2- i)+j+1 for 31> i>= cut_off
step 3: Reassign the remaining next states by placing them in
        the genome as close as possible to the present state.
        Newly assigned states are first processed by step 2. */

line # {
1:      int i, j, k, n, cut_off, new_next_state,
2:          next_step2, last_step2, next_step3, last_step3,
3:          used [max_num_states+2] = {0},

4:          to_be_processed2 [max_num_states],
5:          to_be_processed3 [2*max_num_states] [2];
        /* The to_be_processed arrays are queues which hold the
           states which must be processed by steps 2 & 3. */

        /* STEP 1 */
6:      xchange_states(0, *start_state);
7:      used[1] = 1; /* state 0 is used */
8:      *start_state = 0;
9:      last_step2 = next_step2 = 0;
10:     to_be_processed2[next_step2] = 0;
11:     next_step3 = 0; last_step3 = -1;

        /* STEP 2 */
12:     cut_off = max_num_states/2;
13:     step2(cut_off, &next_step2, &last_step2, &last_step3,
14:          &used[0], &to_be_processed2[0], to_be_processed3);

        /* STEP 3 */
15:     while (next_step3 <= last_step3) {
        /* States appear on this list because the the next state had
           not been reassigned and the desired next state had already
           been used. */

16:         i = to_be_processed3[next_step3] [0];
17:         j = to_be_processed3[next_step3] [1];
18:         ++next_step3;

19:         if (!used[next_state[i][j] + 1]) {

            /* The next state may have been reassigned since this
               state/output pair was put on this list.
               Look for closest available state. */

20:             k = i - 1;
21:             while (used[k+1]) --k; /*at worst will stop at used[0]*/
22:             n = i + 1;

```

Figure 5.6 continued: The SFS Algorithm (continued on next page)

```

23:         while (used[n + 1]) ++n;    /* at worst will stop at
                                         used[max_num_states+1] */
24:         if (((i-k <= n-i) && (k > 0)) || (n == max_num_states))
25:             new_next_state = k;
26:         else
27:             new_next_state = n;

28:         xchange_states(next_state[i][j], new_next_state);
29:         used[new_next_state+1] = 1;
30:         ++last_step2;
31:         to_be_processed2[last_step2] = new_next_state;
32:         step2(cut_off, &next_step2, &last_step2, &last_step3,
33:             &used[0], &to_be_processed2[0], to_be_processed3);
34:     }

35: } /* while */

} /* SFS */

```

Figure 5.6 continued: The SFS Algorithm

The benchmark program can easily be altered to include an SFS reorganization. In addition to adding the procedures in figures 5.1 through 5.6. in the `#define` statements alter the `run_name`, and the file names for `printed_output` (`rlsfs.st` is suggested.) and `perfect_score` (`rlsfs.ps`) to indicate that the SFS algorithm is implemented.

The FSMs are reorganized before parenting the next generation. Since a generation is prepared for parenting in `create_next_generation` (figure 4.22) and `initialize` (figure 4.20), `reorganizeFSM` must be called in these two procedures. The line

```
reorganizeFSM ();
```

is be inserted into `initialize` (figure 4.20) and `create_next_generation` (figure 4.22) immediately after the common line (line 10 of figure 4.20, line 11 of

figure 4.22)

```
row_of_sorted = sort(0, popsize - 1, 0);
```

for the reorganization of the parent pool to be carried out.

There is one other change which is not necessary for proper execution, but can be installed for esthetic purposes. Perfect scoring individuals are printed out before they are reorganized. If it is desirable to see these machines in their reorganized format, **SFS** must be called just before a perfect scoring machine is printed out. To do this insert the line

```
SFS(&start_state);
```

between lines 84 and 85 of **calc_fitness** (figure 4.14). This reorganizes the machine strictly for the printout, but it does not alter its stored format.

Except for the changed title, the output for an SFS run appears the same as for the benchmark. If perfect scoring machines are reorganized for the printout, an observer familiar with the algorithm might recognize a machine as having been reorganized by the SFS algorithm. In the next section two FSMs are walked through an SFS reorganization.

5.1.2 Examples of SFS reorganization

Because the SFS algorithm is rather complex, two examples are presented. The second and longer of the two appears in appendix A. In the examples, in each successive table the changes from the previous table are indicated in boldface. For the first example **SFS** is applied to the FSM in table 4.1. The first step (lines 6-11, figure 5.6) changes the start state, q_{13} , to q_0 changing table 4.1 to table 5.1.

Start state: q0

present state	next state/output for input=0	next state/output for input=1
q0	q5/0	q9/2
q5	q0/2	q5/2
q9	q5/1	q24/2
q24	q0/3	q24/2

Table 5.1: FSM of Table 4.1 after Step 1 of SFS

q_0 is then processed by **step2** (figure 5.5). According to the function, the next state for q_0 with input 0 should be $2 \times 0 + 0 + 1 \rightarrow q_1$, and for q_0 with input 1 the next state should be $2 \times 0 + 1 + 1 \rightarrow q_2$ (line 7, figure 5.5). Since q_5 is the next state for q_0 with input 0 (table 5.1), q_5 must be exchanged with q_1 . Similarly, because q_9 is the next state for q_0 with input 1 (table 5.1), q_9 must be exchanged with q_2 . Making these changes yields table 5.2.

Start state: q0

present state	next state/output for input=0	next state/output for input=1
q0	q1/0	q2/2
q1	q0/2	q1/2
q2	q1/1	q24/2
q24	q0/3	q24/2

Table 5.2: FSM of Table 4.1 after Next States for q_0 Reassigned

At this point states q_0 , q_1 , and q_2 have been reassigned and tagged as **used**, and any transitions to these states will not be changed as **SFS** continues. q_1 and q_2

must now be processed by **step2**. Since the next states for q_1 (q_0 and q_2) have been tagged as used, no further processing is needed for q_1 . Similarly for the next state for q_2 with input 0. But the next state for q_2 with input 1 has to be reassigned. Using the formula (line 7, figure 5.5), q_{24} should become $2 \times 1 + 1 + 1 \rightarrow q_6$. Since q_6 has not been used, the change is made yielding the FSM in table 5.3. q_6 is then tagged as used and processed by **step2**. Because the next states of q_6 have been tagged, the SFS reorganization is complete.

Start state: q_0

present state	next state/output for input=0	next state/output for input=1
q_0	$q_1/0$	$q_2/2$
q_1	$q_0/2$	$q_1/2$
q_2	$q_1/1$	$q_6/2$
q_6	$q_0/3$	$q_6/2$

Table 5.3: FSM of Table 4.1 after SFS

A large FSM is needed to demonstrate step 3 of SFS. To see step 3 applied and to gain more insight into SFS and step 3, see the second example which appears in appendix A. The FSM in appendix A (table A.1) contains 22 states and the SFS reorganization of this FSM is far more complex than that of the previous example.

SFS reorganizes the genome to enhance schemata growth by standardizing transitions to some extent. Referring back to equation 4.2, another way to enhance schemata growth is to reduce the chance that a useful schema will be disrupted by crossover by decreasing the schema's defining length. The MTF algorithm, which is

presented in the next section, moves the relevant data to the front of the genome. This effectively reduces the size of the genome, and hence the defining length of useful schemata.

5.2 The MTF algorithm

The MTF algorithm systematically moves the reachable states of a finite state machine to the front of the genome. As indicated the genome allows for 32 states, but not all machines which are created by the GA utilize all 32 states. By reorganizing an FSM to use states zero through one less than the number of states which can be reached, the relevant data is more closely positioned in the genome. For example the machine in table 4.1 is spread out across the genome. MTF would reorganize this FSM to table 5.4.

Start state: q0

present state	next state/output for input=0	next state/output for input=1
q0	q1/0	q2/2
q1	q0/2	q1/2
q2	q1/1	q3/2
q3	q0/3	q3/2

Table 5.4: FSM of Table 4.1 after MTF

Like SFS, MTF standardizes q_0 as the start state. Recall that the start state appears in the first 5 bits of the genome and each state needs $2 \times (5+2) = 14$ bits to code for its next states (5 bits per state) and outputs (2 bits per output). Therefore,

the machine in figure 4.1 is spread out across the first $5 + 25 \times 14 = 355$ bits of the genome. The equivalent machine in table 5.4 uses only the first $5 + 4 \times 14 = 61$ bits of the genome for the relevant data. The FSM in table 5.4 will obviously have shorter useful schemata. The next section describes the algorithm and its coding. The following section presents a simple example.

5.2.1 MTF: description and coding

The code and description of the MTF reorganization algorithm appear in figure 5.7. The algorithm is very simple. Step 1 is the same as for *SFS*—rename the start state as state 0 (lines 2-3, figure 5.7). The **next_available_state** along the genome is state 1 (line 4). **MTF** moves along the genome, starting at state 0. The **for** loops move the processing through the states and through the inputs for each state (lines 5-6). When state *i* with input *j* is processed, if the next state is past the end of the reorganized part of the genome, the next state is reassigned to the **next_available_state** (lines 7-9), and the **next_available_state** is advanced (line 10). **MTF** does not need a separate queue to retain the states which have to be processed because the table contains that queue. Since each state is moved into the next available position and because this position sequentially moves through the states, the present state part of the transition table actually forms the processing queue. A **used** array is also not needed because all states before the **next_available_state** have been reassigned (used). The looping terminates when all reachable states have been moved to the front of the table (line 5), and hence to the front of the genome.


```

/* ----- */
/*           Move To Front           */
/* ----- */

void MTF (int *start_state)

/* move all reachable states to the front of the genome.
   step 1: make 0 the start state.
   step 2: for each state processed
      a) make its next state on input=0 the next available state
         on the genome, unless it is already a reassigned state.
      b) do the same for input=1. */
{
line #
1:   int i, j, next_available_state;

      /* STEP 1 */
2:   xchange_states(0, *start_state);
3:   *start_state = 0;
4:   next_available_state = 1;

      /* STEP 2 */
5:   for (i=0; i < next_available_state; ++i)
6:     for (j=0; j<2; ++j)
7:       if (next_state[i][j] >= next_available_state) {
8:         xchange_states(next_state[i][j],
9:           next_available_state);
10:        ++next_available_state;
11:      }
} /* MTF */

```

Figure 5.7: The MTF Algorithm

To change the SFS program to MTF, first remove the **SFS** and **step2** procedures, and add the **MTF** procedure. **reorganizeFSM** (figure 5.3) has to be modified to call **MTF** instead of **SFS**. Change the call on line 5 of figure 5.3 from **SFS** to **MTF**, and if **SFS** was called by **calc_fitness** for esthetic printout, then that call also must be changed. Then alter the lines which **#define** the **run_name**, **printed_output** (**rlmtf.st**), and **perfect_score** (**rlmtf.ps**) to indicate that MTF is incorporated into the run, and the MTF program is ready for compilation.

In the next section a simple example of an MTF reorganization is presented.

5.2.2 Example of an MTF reorganization

Because MTF is so simple, a single short example will suffice to demonstrate it. Start with the FSM in table 4.1. The first step of **MTF** (figure 5.7) makes q_0 the start state: that is it interchanges states q_0 and q_{13} (lines 2-3). This changes table 4.1 to table 5.1. This is exactly the same thing SFS does for its first step. The **next_available_state** (line 4) is the state after q_0 , which is q_1 . Starting with q_0 , assign the next state for inputs 0 and 1 to the next two available states (lines 5-10), unless the next states have already been reassigned (line 7). The next state for q_0 with input 0 is q_5 and by MTF should be the **next_available_state**, which is q_1 . Therefore q_5 and q_1 are interchanged (line 8) and the **next_available_state** is incremented to 2 (line 9). Moving along to q_0 with input 1, next state for q_0 with input 1 is q_9 and by MTF should be the **next_available_state**, which is now q_2 . Therefore q_9 and q_2 are interchanged (line 8) and the **next_available_state** is incremented to 3 (line 9). While this process is different from that of SFS, the result at this point is the same (table 5.2).

q_1 is next processed. The next states for q_1 with inputs 0 and 1 are q_0 and q_2 respectively. These states have already been reassigned, indicated by the fact that they are both before **next_available_state** (line 7). Hence no further processing is needed for q_1 .

q_2 is the next state to be processed. The next state for q_2 with input 0 is q_1 , which has already been reassigned (line 7). Therefore no further processing is needed for q_2 with input 0. But the next state for q_2 with input 1 is q_{24} and that is beyond the **next_available_state**, which is now q_3 . Consequently q_3 and q_{24}

are exchanged (line 8) and the `next_available_state` is advanced to q_4 (line 9).

The next state to be processed is q_3 . Both next states, q_0 and q_3 , have been reassigned. Moving on to q_4 for processing halts the looping (line 5). q_4 is not before `next_available_state` (line 5) indicating that all reachable states have been moved to the front of the table, and hence the genome. The final result appears in table 5.4.

Both SFS and MTF could reduce innovation. With more similar appearing machines, crossover is less likely to create something new. Utilizing these algorithms could therefore make the GA more prone to premature convergence. One way to compensate for this is to add a competition between individuals in order to change the static landscape of the fitness function to a landscape which changes with each generation. The next section explains competition and provides the coding to implement it.

5.3 Competition

Premature convergence occurs when the parent pool is overtaken by a suboptimal genome. This occurred in the run which produced the output in figure 5.8. The evolving population quickly attains a maximum fitness of 85. Similar individuals quickly take over a large part of the population and fill up the parent pool.

Base case modelled after Jefferson et al.
 Statistics for run #7 on file r7b.st
 Seed=0.48296342
 Crossover rate=0.01; Mutation rate=0.01
 Population size=65536
 Select the top 3277 individuals or 5.00% of each generation for reproduction.

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
0	78/1	3.7161/ 1.0 / 0 /27008	7.2989
1	70/1	7.8577/ 2.0 / 0 /18385	11.6240
2	70/2	12.1766/ 3.0 / 0 /16026	15.5308
3	70/1	14.7429/ 6.0 / 0 /14862	17.0850
4	74/1	16.7762/ 10.0 / 42 /17745	17.8928
5	80/1	18.3567/ 10.0 / 42 /20133	18.3020
6	80/1	19.2623/ 10.0 / 42 /21130	18.4483
7	79/1	19.8498/ 11.0 / 42 /20784	18.4541
8	79/2	19.4379/ 11.0 / 42 /17181	18.2266
9	80/1	18.6725/ 10.0 / 0 /11011	18.0874
10	80/1	20.6723/ 11.0 / 0 /10427	19.0944
11	81/2	21.8529/ 13.0 / 0 /9569	19.9821
12	81/10	29.9752/ 28.0 / 61 /9611	24.5029
13	82/5	39.2769/ 42.0 / 79 /9849	28.0425
14	82/35	48.3407/ 47.0 / 79 /26073	31.7393
15	83/1	49.7960/ 58.0 / 80 /30219	32.0161
16	83/1	49.7920/ 53.0 / 80 /26615	31.9080
17	84/1	48.9350/ 43.0 / 82 /14444	31.4711
18	84/4	54.1667/ 79.0 / 82 /28899	31.2277
19	84/16	54.7868/ 79.0 / 82 /29726	31.1520
20	84/66	54.9068/ 79.0 / 82 /29851	31.0771
21	84/227	54.4014/ 79.0 / 82 /29269	31.2023
22	84/812	52.8376/ 78.0 / 82 /26664	31.6384
23	85/8	47.8541/ 42.0 / 82 /16200	32.2410
24	86/1	45.6152/ 42.0 / 84 /18487	32.3465
25	85/106	46.9562/ 42.0 / 84 /19711	32.4502
26	85/403	48.0203/ 42.0 / 84 /20296	32.4963
27	85/1860	49.4446/ 42.0 / 84 /19561	32.4576
28	85/9925	49.1650/ 42.0 / 85 /9925	32.4568
29	85/20745	49.8664/ 42.0 / 85 /20745	32.7179
30	85/21926	50.9184/ 44.0 / 85 /21926	32.6939
31	85/23690	52.2127/ 50.0 / 85 /23690	32.6795
32	85/25027	53.2120/ 59.0 / 85 /25027	32.5720
33	85/25847	53.8965/ 78.0 / 85 /25847	32.5748
34	85/26472	54.3810/ 78.0 / 85 /26472	32.5290
35	85/26555	54.4535/ 78.0 / 85 /26555	32.5426
36	85/26796	54.4999/ 78.0 / 85 /26796	32.5788
37	85/26954	54.6478/ 78.0 / 85 /26954	32.5519
38	85/26900	54.5289/ 78.0 / 85 /26900	32.5846
39	85/27154	54.8597/ 79.0 / 85 /27154	32.5391
40	85/27285	54.8983/ 79.0 / 85 /27285	32.5821
41	85/27066	54.6912/ 78.0 / 85 /27066	32.5471
42	85/26842	54.4200/ 78.0 / 85 /26842	32.7018
43	85/26992	54.4673/ 78.0 / 85 /26992	32.6669
44	85/27320	54.7612/ 79.0 / 85 /27320	32.6243
45	85/27251	54.7477/ 79.0 / 85 /27251	32.6546
46	85/27266	54.7601/ 79.0 / 85 /27266	32.6276
47	85/26963	54.2763/ 78.0 / 85 /26963	32.7590
48	85/26881	54.3092/ 78.0 / 85 /26881	32.7520
49	85/27326	54.9054/ 79.0 / 85 /27326	32.6446
50	85/27151	54.5582/ 78.0 / 85 /27151	32.7468

Figure 5.8: Example of Premature Convergence (continued on next page)

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
51	85/27275	54.5559/ 79.0 / 85 /27275	32.7504
52	85/27039	54.4343/ 78.0 / 85 /27039	32.8020
53	85/26778	54.2419/ 78.0 / 85 /26778	32.8035
54	85/27161	54.6802/ 79.0 / 85 /27161	32.7173
55	85/27032	54.4643/ 78.0 / 85 /27032	32.7540
56	85/27240	54.3823/ 78.0 / 85 /27240	32.8372
57	85/27065	54.5332/ 79.0 / 85 /27065	32.7871
58	85/27053	54.3053/ 78.0 / 85 /27053	32.7971
59	85/27065	54.3079/ 78.0 / 85 /27065	32.8426
60	85/26866	54.1505/ 78.0 / 85 /26866	32.9019
61	85/27027	54.2435/ 78.0 / 85 /27027	32.8454
62	85/27061	54.1236/ 78.0 / 85 /27061	32.9368
63	85/26978	54.0425/ 78.0 / 85 /26978	32.9256
64	85/26922	54.1561/ 78.0 / 85 /26922	32.9152
65	85/27270	54.3465/ 79.0 / 85 /27270	32.9389
66	85/26993	54.0707/ 78.0 / 85 /26993	32.9375
67	85/27078	54.1169/ 78.0 / 85 /27078	32.9898
68	85/27173	54.1822/ 78.0 / 85 /27173	33.0029
69	85/27134	54.3050/ 78.0 / 85 /27134	32.9180
70	85/27175	54.3922/ 79.0 / 85 /27175	32.9406

Figure 5.8 continued: Example of Premature Convergence

There are several situations which can prevent the GA from moving the population ahead. Recall that the parents are selected from the top $\text{select}=3277$ members of a generation. If these machines have the same or very similar genomes, crossover has little, if any, chance of creating something new. If these machines are small, say 10 states, only $5 + 10 \times 14 = 145$ bits of the genome carry relevant data. With a mutation rate of 1%, one or two of these bits (0.01×145) can be expected to be changed by mutation. Most of these changes will reduce the fitness of the individual. For example, one run which was permitted to breed 70 generations, even though the first perfect scoring individual appeared in generation 32, produced a set of machines whose similarities are represented in table 5.5. The table represents a schema shared by the ants. For one FSM, *** is $q_0/0$ yielding an ant which takes 195 time steps to traverse the trail. When *** is $q_7/2$, $q_0/2$, or $q_4/2$ the ant needs 193 time steps. Other relatives which appeared were $q_5/2$ for *** requiring 197 time steps, and $q_6/3$

needing the full 200 time steps to traverse the trail. The success of this set of ants, differing in only a single transition/output, might suggest that, for some inexplicable reason, the GA is altering only this part of the genome. In fact this example indicates that this set of changes does not destroy the fitness: the numerous other machines which had other changes in the genome had a lower fitness.

present state	next state/output for input=0	next state/output for input=1
q0	q1/1	q2/2
q1	q3/1	q4/2
q2	q5/2	q1/2
q3	q5/0	q2/2
q4	q5/2	q6/2
q5	q7/2	***
q6	q6/0	q1/2
q7	q/0	q6/2

Table 5.5: Template for a Family of Ants with Differences in One Transition/Output

Another run, which was also permitted to breed 70 generations, even though a perfect scorer appeared in generation 38, produced the set of relatives whose similarities are represented in table 5.6. When ### is replaced with q₁/0 and @@@ is replaced with q₇/2 it takes the machine 187 time steps to cover the 89 trail steps. When @@@ in the previous machine was changed to q₃/2, probably as the result of mutation of a single bit, the machine still scores perfectly, but this machine needs 191 time steps to cover the trail. When ### in the second machine was

changed to $q_0/3$, the fitness and time remain unchanged. The changing of $q_1/0$ (which appears as 0000100 in the genome) to $q_0/3$ (which appears as 0000011 in the genome) could have occurred as the result of mutating three adjacent bits, but it is more likely to have occurred as the result of a crossover. The first of these three machines must have produced many progeny, but only a few different ones attained the optimal fitness.

present state	next state/output for input=0	next state/output for input=1
q0	q1/0	q2/2
q1	q3/1	q4/2
q2	###	q3/2
q3	q5/1	q6/2
q4	q6/2	q3/2
q5	q7/0	q8/2
q6	q3/2	q2/2
q7	q0/2	q7/2
q8	q7/2	@/@/@

Table 5.6: Template for a Few Perfect Scoring Ants with Differences in Two Transitions/Outputs

When relevant data does not fill every bit in the genome, fewer bits of relevant data will be mutated. When crossover and/or mutation create a new phenotype, the fitness of the newborn can be lower than that of the parents. Note that in figure 4.25 the average fitness levels off around generation 26. Also look at the mode and number of individuals with that fitness. The parent pool (consisting of the top 3277 ants) does not have individuals with a fitness less than the mode.

Therefore starting at generation 19 (figure 4.25), parents have a fitness of at least 80. A high fitness fills up a large portion of each generation (between 12.5% and 43%) as indicated by the number of individuals with mode. Also there are a significantly lower average and median for each of these populations. This indicates that parents with a high fitness are producing low-scoring children. In addition there is probably a reduction of innovation due to a parent pool consisting of similar genomes. Consequently, the high-scoring children which will form the next parent pool may not be significantly different from their parents.

Base case modelled after Jefferson et al.
 Statistics for run #30 on file r30b.st
 Seed=0.85823880
 Crossover rate=0.01; Mutation rate=0.01
 Population size=65536
 Select the top 3277 individuals or 5.00% of each generation for reproduction.

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
0	58/7	3.7190/ 1.0 / 0 /27247	7.3241
1	70/1	7.8323/ 2.0 / 0 /18569	11.6077
2	78/1	11.7236/ 3.0 / 0 /16270	15.2258
3	78/2	14.1280/ 5.0 / 0 /15171	16.7544
4	78/4	16.1601/ 10.0 / 42 /16003	17.5561
5	81/1	17.7928/ 10.0 / 42 /18489	18.0866
6	81/1	18.3326/ 10.0 / 42 /18494	18.1056
7	80/2	18.3824/ 10.0 / 42 /16674	17.9642
8	81/2	17.8925/ 10.0 / 42 /11839	17.7116
9	81/1	17.9904/ 10.0 / 0 /10848	17.9229
10	81/1	18.1175/ 10.0 / 0 /9891	17.7954
11	81/1	19.5357/ 11.0 / 0 /9016	18.3897
12	81/3	20.8326/ 12.0 / 0 /8508	19.2128
13	81/7	23.0388/ 16.0 / 0 /8349	20.6642
14	81/14	25.3135/ 21.0 / 58 /9872	21.9316
15	81/39	27.7204/ 27.0 / 58 /11878	22.9996
16	82/1	28.8656/ 27.0 / 42 /9288	24.5009
17	82/1	41.4281/ 42.0 / 80 /18567	30.4038
18	82/5	44.9957/ 42.0 / 80 /17054	30.8681
19	82/2	49.6899/ 46.0 / 81 /24185	31.0441
20	82/1	53.5471/ 80.0 / 81 /29200	30.4957
21	82/4	55.3736/ 80.0 / 81 /31783	30.1042
22	82/20	56.3548/ 81.0 / 81 /33363	29.9466
23	82/57	56.3084/ 81.0 / 81 /33504	29.8992

Figure 5.9: Example of an Unsuccessful Run (continued on next page)

Gen	Max/# with Max	Average/Median/Mode/# with Mode	Standard dev.
24	82/187	56.5181/ 81.0 / 81 /33705	29.9683
25	84/1	56.1573/ 81.0 / 81 /32553	29.9602
26	84/2	55.3858/ 80.0 / 81 /28704	29.8535
27	84/6	52.9048/ 56.0 / 81 /13161	29.5627
28	84/7	52.2641/ 52.0 / 82 /19673	29.3857
29	84/22	52.4103/ 52.0 / 82 /19895	29.5209
30	84/111	51.5901/ 49.0 / 82 /17933	30.0348
31	84/531	50.6058/ 47.0 / 83 /12593	31.0139
32	84/2495	55.2657/ 79.0 / 83 /25126	30.0359
33	85/1	53.1192/ 52.0 / 84 /15929	29.7098
34	85/3	54.3094/ 56.0 / 84 /23657	29.7017
35	85/10	54.6090/ 57.0 / 84 /23808	29.6042
36	85/41	54.8912/ 58.0 / 84 /24131	29.5418
37	85/169	55.0085/ 58.0 / 84 /24292	29.5680
38	85/730	54.9841/ 58.0 / 84 /23356	29.5169
39	86/1	54.4352/ 55.0 / 84 /19191	29.4439
40	86/5	54.1050/ 53.0 / 85 /20799	29.6215
41	86/20	54.3600/ 53.0 / 85 /21567	29.6883
42	86/52	54.4796/ 53.0 / 85 /21663	29.7553
43	86/159	54.5285/ 53.0 / 85 /21619	29.7139
44	86/503	53.9719/ 53.0 / 85 /20668	29.7487
45	86/1697	52.9792/ 52.0 / 85 /18922	29.9071
46	86/7237	51.1180/ 52.0 / 85 /11591	29.9374
47	86/17496	52.2747/ 52.0 / 86 /17496	29.8381
48	86/17475	52.1763/ 52.0 / 86 /17475	29.8532
50	86/17677	52.5339/ 52.0 / 86 /17677	29.7720
51	87/1	52.6176/ 52.0 / 86 /17941	29.8655
52	86/18037	52.8100/ 52.0 / 86 /18037	29.8821
53	86/18037	52.6003/ 52.0 / 86 /18037	29.9441
54	87/1	52.4585/ 52.0 / 86 /17989	29.9091
55	87/1	52.6242/ 52.0 / 86 /18033	29.9459
56	86/18323	52.7689/ 52.0 / 86 /18323	29.9270
57	87/1	52.3235/ 52.0 / 86 /17822	29.9750
58	86/18090	52.5240/ 52.0 / 86 /18090	29.9793
59	86/17965	52.5231/ 52.0 / 86 /17965	29.9541
60	86/18146	52.6640/ 52.0 / 86 /18146	29.9816
61	87/1	52.5207/ 52.0 / 86 /18011	29.9798
62	87/1	52.5978/ 52.0 / 86 /18110	29.9897
63	87/1	52.5226/ 52.0 / 86 /18047	30.0488
64	86/18158	52.5684/ 52.0 / 86 /18158	30.0247
65	86/18203	52.5325/ 52.0 / 86 /18203	30.0613
66	86/17987	52.5670/ 52.0 / 86 /17987	29.9707
67	86/18248	52.6566/ 52.0 / 86 /18248	30.0245
68	86/18110	52.4253/ 52.0 / 86 /18110	30.0879
69	87/1	52.0291/ 52.0 / 86 /17652	30.0739
70	87/1	52.3869/ 52.0 / 86 /18035	30.0552

Figure 5.9 continued: Example of an Unsuccessful Run

Another run (figure 5.9) shows a different pattern. Note that FSMs with a fitness of 87 intermittently appear, but are unable to maintain a continuous presence. Apparently there is not enough schema support in the parent pool to create enough

children with a more useful schema, which in turn would move the population to a new area of the genome space. While the run in figure 5.9 might move ahead if it were permitted to continue beyond generation 70, the run in figure 5.8 appears to be remaining at a plateau. Adding a few individuals with a lower fitness into the parent pool can dramatically alter the results by creating more innovation. Introducing a competition to determine an individual's fitness can let some less-fit individuals, less-fit based on trail following ability, into the parent pool. This in turn can push the population into another portion of the genome space.

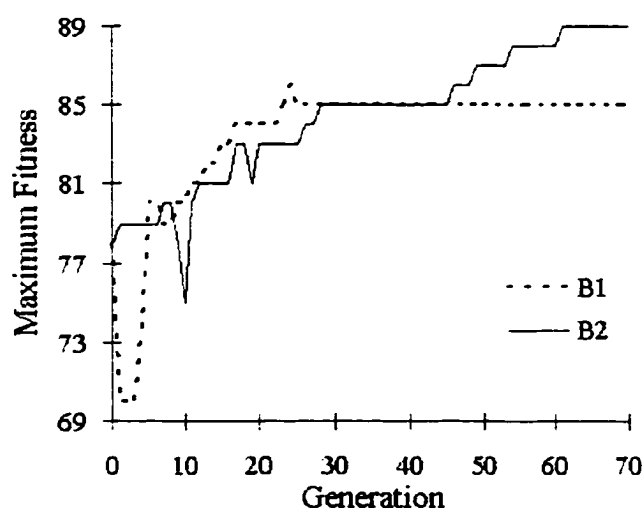


Figure 5.10: Comparison of Runs with and without Competition

generation	0	1	2	3	4	5	6	7	8	9	10	11	12	13
max. fitness B1	78	70	70	70	74	80	80	79	79	80	80	81	81	82
max. fitness B2	78	79	79	79	79	79	79	80	80	78	75	80	81	81

Table 5.7: Comparison of Runs with and without Competition (continued on next page)

generation	14	15	16	17	18	19	20	21	22	23	24	25	26	27
max. fitness B1	82	83	83	84	84	84	84	84	84	85	86	85	85	85
max. fitness B2	81	81	81	83	83	81	83	83	83	83	83	83	84	84

generation	28		45	46	47	48	49		53	54	...	60	61	...	70
max. fitness B1	85		85	85	85	85	85		85	85	...	85	85	...	85
max. fitness B2	85		85	86	86	86	87		87	88	...	88	89	...	89

Table 5.7 continued: Comparison of Runs with and without Competition

Competition has been widely used in evolutionary simulations. (For example Fogel 1991, Angeline and Pollack 1993, Fogel 1993, Angeline 1994, and Reynolds 1994 to name a few.) In Figure 5.10, the dotted line labelled B1 is the run in figure 5.8. The solid line labelled B2 on the same graph added competition to the B1 run—everything else is the same including the initial population. The same data comparing these two runs appears in table 5.7. The progress of the B2 run (with competition) is erratic at the beginning, and for the most part tends to lag behind the B1 run (without competition). Both runs appear to prematurely converge to a fitness of 85—B1 in generation 23 and B2 in generation 28. But in generation 46 B2 moves off of what appears to be a local optimum and continues on to find a solution in generation 61.

To add competition to the three programs (the benchmark, and the versions modified for SFS and MTF), add

```
#define num_competitions 10
```

to the **#define** statements at the beginning of the program (section 4.3). **num_competitions** is the number of competitions that each individual will face. In this implementation, each individual faces 10 randomly selected competitors. The files to receive the output should also be changed (perhaps by adding **_c** to the file names) to indicate that fitness is determined using competition.

```

/* ----- */
/*           Competition           */
/* ----- */

void compete(void)
/* holds a competition between an individual and a random opponent.
   the altered fitness favors those with most wins. */
{
line #
1:   int i, j, k, wins[popsize]={0};

2:   for (i=0; i<popsize; ++i)
3:     for (j=0; j<num_competitions; ++j) {
4:       k = Rnd(0, popsize-1);
5:       if (fitness[k] < fitness[i])
6:         wins[i] += 2;
7:       else
8:         if (fitness[k] == fitness [i])
9:           ++wins[i];
10:    }

11:   for (i=0; i<popsize; ++i)
12:     fitness[i] += 100*wins[i]; /* bonus points for winning */

} /* compete */

```

Figure 5.11: Competition Procedure

The results of the competition will be used to alter the **fitness**. The **compete** procedure (figure 5.11) carries out the competition and alters the **fitness** to include the results of the competition. If a competitor covered fewer trail steps, the individual is credited with two points for the competition (lines 5-6).

If the competitor traversed more trail steps, the individual receives no credit for the competition. When there is a tie, the individual receives a single point for the competition (lines 7-9). The points for the ten competitions are accumulated (lines 6 and 9). The new **fitness** is the sum of the number of trail steps traversed and 100 times the points earned from the competitions (lines 11-12).

The competition cannot be carried out till the trail following ability (calculated in **calc_fitness**, figure 4.14) has been determined for the entire population, and **compete** must be invoked before the population is sorted according to the competition-based **fitness**. **compete** is invoked by **create_next_generation** (figure 4.22) and **initialize** (figure 4.20) as part of the process of preparing a population for parenting. Therefore add the line

```
compete();
```

just before the line

```
row_of_sorted = sort(0, popsize - 1, 0);
```

which appears at the end of both **create_next_generation** (line 11, figure 2.22) and **initialize** (line 10, figure 2.20).

The procedure which calculates the statistics for the file with the **st** extension must also be changed. This is necessary because the statistics are based on the ants' ability to traverse the trail, but a population whose fitness is determined by competition is not ordered by trail following ability. Since the original **save_stats** subroutine (figure 4.21) assumes the data is ordered by trail following ability, it has to be modified. Also each ant's **fitness** includes the result of the competition as the primary determination of fitness. This must be removed for the calculations of

average and standard deviation which are to be based on trail following ability. The new **save_stats** subroutine is listed in figure 5.12. Since the primary determination of fitness is the competition, an FSM which covered the most trail steps is not necessarily the first on the sorted list. In figure 4.21 (line 3) the maximum number of trail steps covered by an individual in the population is determined by the fitness of the first individual on the sorted list. **fitness_count[i]** indicates the number of FSMs that covered *i* trail steps. Therefore the maximum number of trail steps covered by an individual in the population is also the largest *i* such that **fitness_count[i] ≠ 0** (line 4, figure 5.12). The median is no longer at the center of the sorted list (lines 7-12, figure 4.21). Here also **fitness_count[i]** is used. The **while** loop (lines 9-12, figure 5.12) permits *j* to accumulate the number of individuals till it has counted more than halfway through the population. *i* indicates the fitness of the *j*th individual. If **popsize** is odd or if *j* is past the midpoint of the population (lines 13-14, figure 5.12), *i* is the median. Otherwise find the next higher fitness and average the two (lines 15-19, figure 5.12). The computation of the average (lines 20-22, figure 5.12) and standard deviation (lines 23-25, figure 5.12) must isolate (lines 21 and 24, figure 5.12) the trail following ability (the least two significant digits of the **fitness**) when competition has been included in the **fitness**. The location of the mode(s) is identical in the two subroutines (figures 4.21, and 5.12), as is the printing out of the data. **save_stats** of figure 5.12 replaces the beginning of the one in figure 4.21.

```

/* ----- */
/*           Save Statistics           */
/* ----- */

void save_stats(void)
{
line #
1:   int max = maxfitness, num_of_max, mode=maxfitness, multimode=-1,
2:       i, j, k, fitness_sum=0;
3:   double avg=0, median, standard_dev=0;

/* ----- */
/*           Determine maximum         */
/* ----- */

4:   while (!fitness_count[max]) --max;
5:   num_of_max = fitness_count[max];

/* ----- */
/*           Determine median          */
/* ----- */

6:   k = popsize/2;
7:   if (2*k != popsize) ++k;
8:   i = -1; j = 0;
9:   while (j < k) {
10:      ++i;
11:      j += fitness_count[i];
12:   }
13:   if ( j>k || 2*k>popsize)
14:      median = (double) i;
15:   else {
16:      j = i+1;
17:      while (!fitness_count[j]) ++j;
18:      median = (double) (i+j)/2.0;
19:   }

/* ----- */
/*           Determine average         */
/* ----- */

20:  for (i=0; i<popsize; ++i)
21:      avg += (double) (fitness[i] % 100);
22:  avg /= (double) popsize;

/* ----- */
/*           Determine standard dev.   */
/* ----- */

23:  for (i=0; i<popsize; ++i)
24:      standard_dev += pow((double) (fitness[i]%100) - avg, 2.);
25:  standard_dev = sqrt(standard_dev/ (double) popsize);

```

Figure 5.12: The Procedure to Calculate and Output the Statistics for Each Generation: Modified for Competition (continued on next page)

```

/* ----- */
/*           Determine mode           */
/* ----- */

:

} /* save_stats */

```

Figure 5.12 continued: The Procedure to Calculate and Output the Statistics for Each Generation: Modified for Competition

In addition, because the population is not ordered by trail following ability, the termination test for the **for** loop in the **main** procedure must be changed. The loop in the **main** procedure of the program which does not contain competition (lines 6-9, figure 4.16) is

```

for (generation=1;
    (generation <= last_generation)
    && (fitness[sorted[row_of_sorted][0]] < maxfitness);
    ++generation)

```

This will change to

```

for (generation=1;
    (generation <= last_generation)
    && (fitness_count[maxfitness] == 0);
    ++generation)

```

in the program which includes competition. Note that the only change is in the third line. Instead of checking to see if the first individual on the sorted list has the **maxfitness** based on trail following ability, the test checks to see if any individual has that **maxfitness**.

The modifications for competition can be added to the three previous programs—the benchmark, and the versions modified for SFS and MTF. These six programs form the base for most of the experimentation.

The major goal of this study is to determine whether SFS and/or MTF will

speed up the evolutionary process. To counter the possibility that these two operators might cause frequent premature convergence, experiments were performed for all methods both with and without competition. The programs, which produced the data for the evaluation of the effect of SFS and MTF on the efficiency of the GA, also recorded solutions to the trail problem. Several of these were considered particularly "interesting," and a few of them are presented in the next chapter. Then in chapter 7 the different evaluation criteria are developed. The evaluation of the data obtained from the experimentation follows in chapter 8.

>>>●<<<

Chapter 6

Interesting ants bred during experimentation

During testing some interesting FSMs were observed. Families of successful FSMs have already been mentioned in section 5.3 and highlighted in tables 5.5 and 5.6. These two families are interesting from an evolutionary standpoint. As pointed out in section 5.3, these two families indicate that only a few variations of a genome bred by this GA retain a high fitness. They also demonstrate that there are viable variations of a genome; that is a variation which retains or perhaps can exceed the fitness of its parents. Both individual ants and families are presented in this chapter, but the focus here is the relevance to the trail following problem.

Jefferson et al. (1992) presented Champ-100. Champ-100 is a 13-state FSM which takes the full 200 time steps to complete the trail. FSMs which take less time and have fewer states are considered interesting. Not all successful FSMs were recorded during our testing—only the first ten successful ants in a generation. Consequently the observations do not necessarily indicate the smallest FSM nor the minimum number of time steps needed to traverse the trail.

The examples in this chapter are presented with decreasing number of states and generally (but not always) with increasing time steps. All of the FSMs presented,

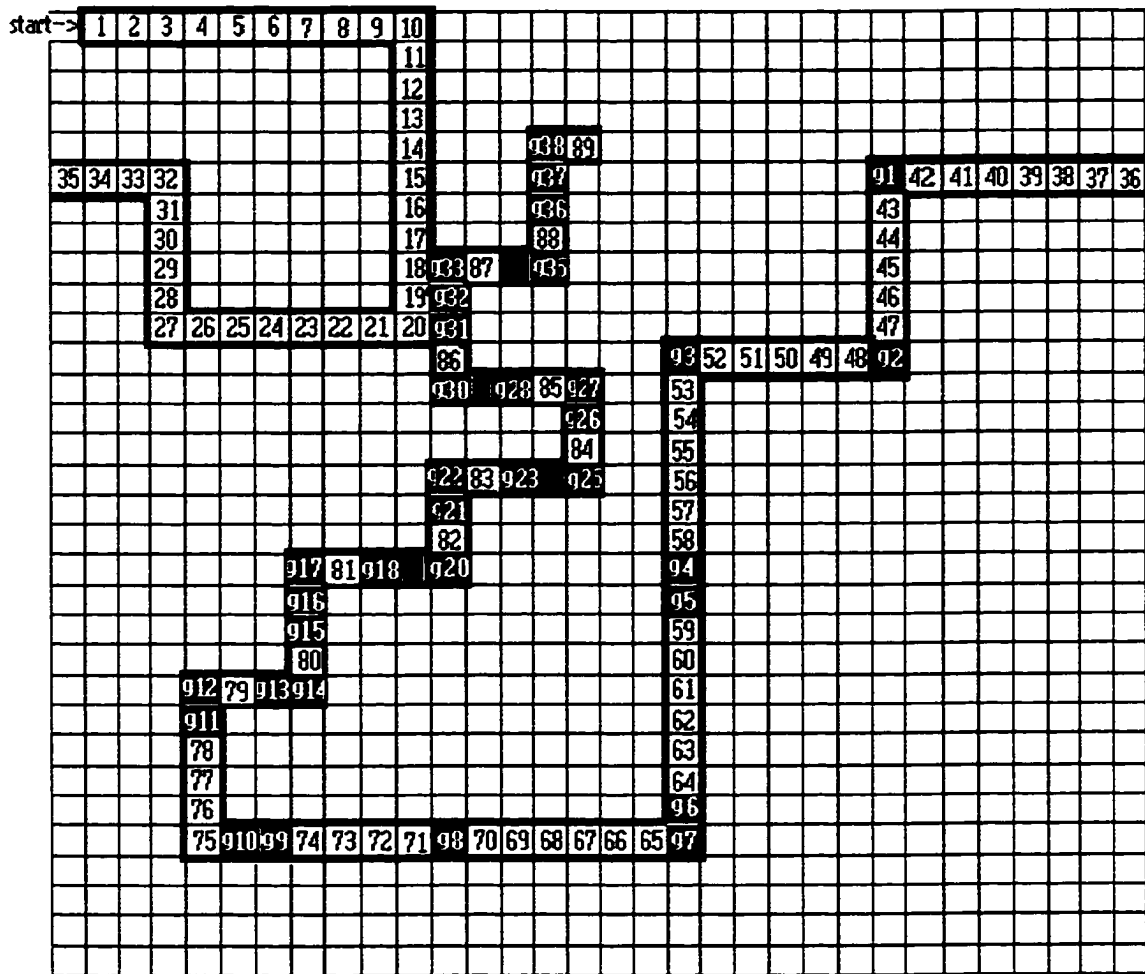


Figure 6.1: The John Muir Trail with Marked and Unmarked Trail Steps Labelled
(Printed with permission of Perseus Books, LLC)

except one, come from some form of MTF run. This is not unexpected since theoretically the MTF operator should encourage smaller FSMs. In addition, all the FSMs which are the result of an MTF run have a start state of 0. There is no pressure in the fitness function or according to schema theory to encourage

minimization of the time steps needed to traverse the trail. The FSMs are presented exactly as they are printed out by the GA. Recall that an output of 0 indicates a right turn; 1, a left turn; 2, take one step straight ahead; and 3, do nothing. An input of 1 indicates that the next trail step ahead is marked. Note that in all the examples, an output of 2 is the prevalent response, at times the only response when the input is 1. Also note that an output of 3 is rare in the examples presented. The rendition of the John Muir trail in figure 6.1 includes labels of g_i for the unmarked trail steps. This is done to facilitate the trail traversals presented in this chapter. In several of the examples, a sequence of five states is presented during which time an ant is turning in place to look around when the trail is not in evidence. Note that right and left turns are sometimes navigated with a subsequence of the 5-state sequence. Hence, the 5-state sequence is not completed when a marked step is in front of an ant after one of the turns in the sequence.

6.1 Best observed time: 181 time steps

A family of 12-state FSMs covered the 89 trail steps in 181 time steps. The family is represented by the template in table 6.1. Table 6.2 indicates four of the successful variations. Figure 6.2 represents the first ant in table 6.2. When this FSM is applied to the John Muir trail the transitions for states 1 and 5 with input 1 are never used. When these two transitions and their corresponding outputs are eliminated, this FSM can then be reduced to an 11-state equivalent by combining states 5 and 9 into a single state.

Start state: 0

present state	next state/output for input=0	next state/output for input=1
0	1/0	2/2
1	3/0	###
2	4/2	3/2
3	5/0	6/2
4	7/2	8/2
5	8/2	9/1
6	5/2	9/2
7	3/1	10/2
8	0/0	11/2
9	@@@	7/2
10	9/2	8/2
11	2/2	4/2

Table 6.1: Template for 12-state Perfect Scoring Ants with Time=181

###	@@@
1/3	8/2
4/2	8/1
0/1	6/2
4/2	4/2

Table 6.2: Substitutions for ### and @@@ in Table 6.1

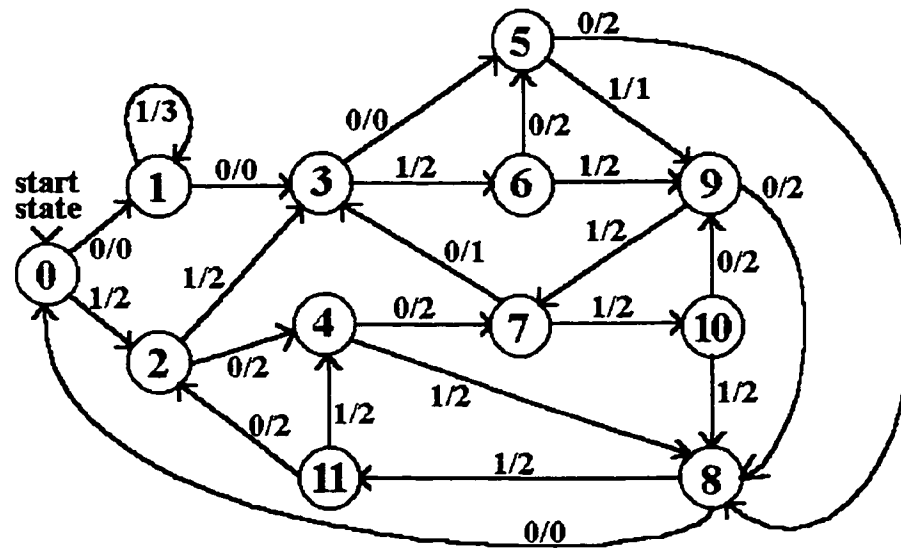


Figure 6.2: 12-state Perfect Scoring Ant with Time=181

Referring to figure 6.1, when an ant is on any one of the marked steps 10, 20, 27, and 75, or unmarked steps g2, g7, g12, g17, g22, g30, g33, and g38, the ant is initially in state 8. An input of 0 results in a right turn, thus putting the ant immediately back on track. An ant in state 7 will make a left turn when the trail disappears. This takes an ant through the left turns at marked step 32, and unmarked steps g1, g14, and g35. The sequence of states 8-0-1-3-5 yields a 360° clockwise turn when there is no marked step in any of the adjacent boxes. Recall that the marking is removed from a step once an ant moves onto it in order to prevent backtracking on the trail. This 360° clockwise turn, or parts of it are used at marked step 47, and unmarked steps g3, g16, g20, g25, g27, g29, and g37. Note

that the first two states of the sequence are used for right turns. The state sequences 10-9-8 and 6-5-8 can take an ant across two unmarked steps but only the 6-5-8 sequence is used by the ant on this trail. It takes the ant from marked step 78 to g12, from 80 to g16, from 82 to g22, from 84 to g27, from 85 to g29, and from 88 to g37. The sequence of states 11-2-4-7 can take an ant across three unmarked steps, but it is never utilized on the John Muir trail. The shorter subsequences 11-2-4 and 2-4-7 are used to take an ant from marked step 58 to g5 (11-2-4), from 74 to g10 (11-2-4), from 79 to g14 (2-4-7), from 81 to g19 (2-4-7), from 83 to g24 (2-4-7), from 86 to g32 (2-4-7), and from 87 to g35 (2-4-7).

The state sequence 7-3-5 (used at steps 52, g6, g19, g24, and g32) has an ant looking only to the left for the missing trail before stepping ahead in the original direction. This saves time, but if it is utilized at the wrong location it could take the ant off the trail and result in aimless wandering to try to relocate the trail. Since the latter does not occur when this FSM is applied to the John Muir trail, it helps keep the time down. It might seem plausible to stay in a single state for a marked straightaway, but this is not the case. For example, to get from the start position to marked step 10, an ant goes through the state sequence 0-2-3-6-9-7-10-8-11-4-8. The ant cycles through the sequence 8-11-4 on a marked straightaway.

In the next section some smaller FSMs with higher traversal times are presented.

6.2 Intermediate FSMs

The FSMs presented in this section have neither the best time nor the fewest states, but the combination of time and number of states is notable. The 11-state FSM in figure 6.3 takes 188 time steps to traverse the trail. The blank positions in the table are never used which is why they have been omitted.

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/0	2/2
1	3/0	3/2
2	4/2	5/2
3	6/0	7/2
4	8/2	—
5	0/0	5/2
6	5/2	1/2
7	6/2	9/2
8	3/1	—
9	10/2	9/2
10	1/2	1/3

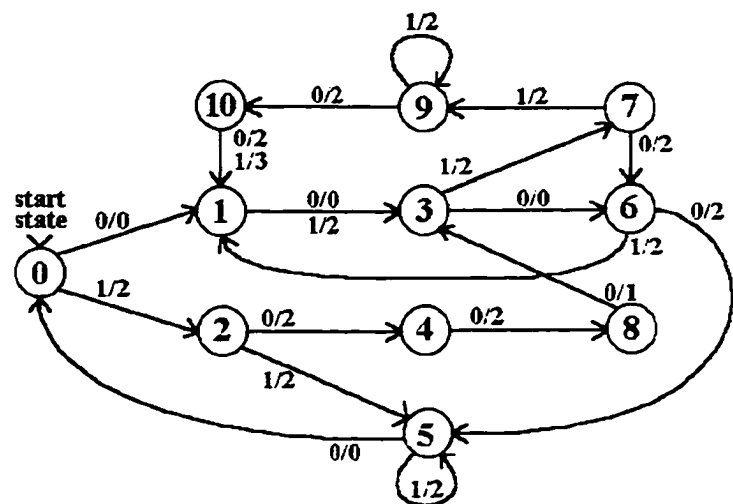


Figure 6.3: 11-state Perfect Scoring FSM with Time=188

This FSM is of interest because at one point it takes the ant off the trail. Instead of wandering aimlessly after that, the ant backtracks onto an unmarked step

and continues on to complete the trail before its time is up. When the ant is on step 42, it is in state 9 facing g1, and receiving an input of 0 from g1. This brings the ant to state 10 on g1 with an input of 0 from the square to the left of g1. The ant moves onto the square to the left of g1 and into state 1. The ant is no longer on the trail. During the next two time steps with inputs of 0, the ant makes two right turns and moves into state 3, followed by state 6. The ant is now facing g1 standing on the square to the left of g1, and in state 6 receiving an input of 0. This brings the ant back onto g1, and into state 5. Step 42 is now directly in front of the ant, but since the ant was previously on step 42 the marking was erased to prevent backtracking on the trail. Consequently, the ant on g1 facing step 42 receives an input of 0. The ant makes a right turn and moves into state 0. At this point the ant is on g1, facing marked step 43, and therefore receiving an input of 1. This takes the ant onto step 43 and back onto the marked trail. It is this recovery which makes the FSM of figure 6.3 so interesting.

With both fewer states and better time, the 10-state FSM in figure 6.4 requires 185 time steps to traverse the 89 marked trail steps. The blank position for state 9 with input 1 contained different values during the run, but these values are irrelevant since an ant on the trail in state 9 never receives an input of 1. Note that the only response to a marked trail step directly ahead (that is to an input of 1) is to move ahead onto that marked step. Also note that there is no output of 3. That is an ant always takes some action; a time step is never used strictly to change state.

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/2	1/2
1	2/0	3/2
2	4/0	5/2
3	6/2	0/2
4	7/0	8/2
5	9/2	7/2
6	0/2	4/2
7	0/0	6/2
8	7/1	0/2
9	8/2	—

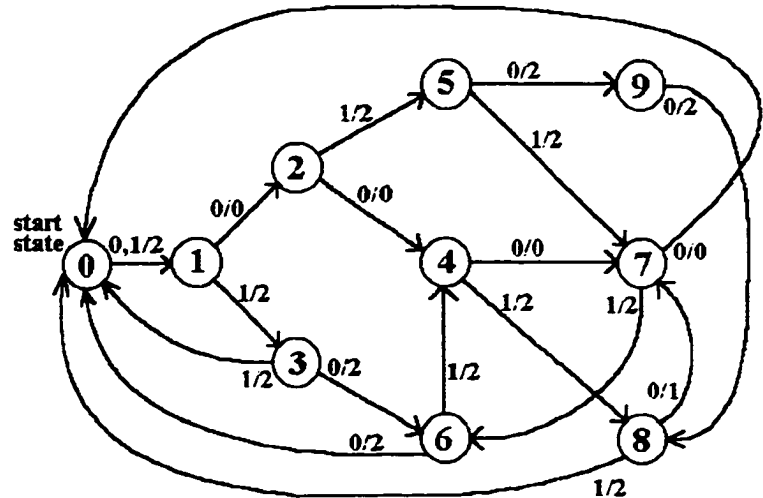


Figure 6.4: 10-state Perfect Scoring FSM with Time=185

To walk from step 10 to step 20, the state sequence is 2-5-7-6-4-8-0-1-3-0-1. Subsequences of this, and the cycle 3-0-1 are utilized along marked straightaways. All right turns are efficiently handled by state sequence 1-2. The left turns at steps 32, g14, and g35 are accomplished with state sequence 8-7. The sequence of states 1-2-4-7-0 takes the ant through a 360° clockwise turn when no marked trail steps are in sight. All or part of this sequence is invoked at steps g1, 47, g3, g6, g16, g20, g25, g27, g29, and g37. While the sequence 3-6-0-1 can take an ant from a marked step across three successive unmarked steps, this full sequence is not used in the trail traversal. The subsequences 3-6-0 and 6-0-1, along with 5-9-8 are used to cover the

two unmarked steps following steps 58 (3-6-0), 74 (3-6-0), and 78 (6-0-1), 79 (5-9-8), 80 (6-0-1), 81 (5-9-8), 82 (6-0-1), 83 (5-9-8), 84 (6-0-1), 85 (6-0-1), 86 (5-9-8), 87 (5-9-8), and 88 (6-0-1).

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/0	2/2
1	3/1	1/2
2	1/0	3/2
3	5/1	6/2
4	6/2	3/2
5	7/0	8/2
6	3/2	2/2
7	0/2	7/2
8	7/2	7/2

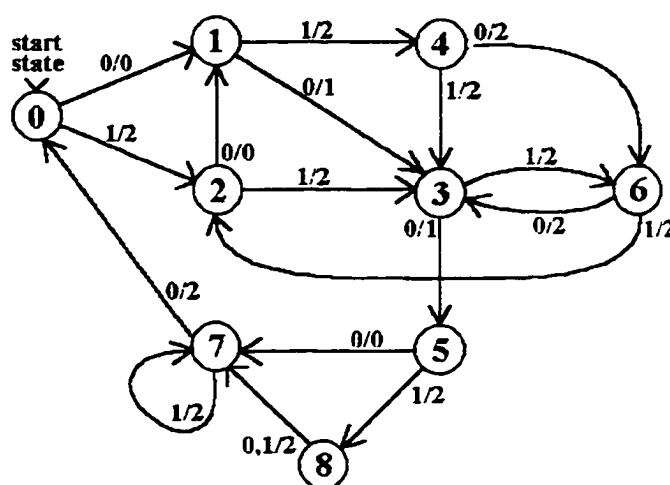


Figure 6.5: 9-state Perfect Scoring FSM with Time=187

The FSM in figure 6.5 takes 187 time steps to complete the trail. This ant has two interesting relatives. Both relatives take 4 time steps longer to cover the trail. For one relative, present state 2 with input 0 moved the FSM to state 0 and output a 3. This simply wastes a time step whenever the FSM is in state 2 and receiving an input of 0. This occurs on steps 10, 20, 27, and 75 explaining the additional 4 time steps that this ant uses. For the other relative, the only part of figure 6.5 which is

affected is the next state for state 8 with an input of 1. For this relative the next state for state 8 with input 1 is 3 instead of 7. This causes this relative to use four additional time steps on step 42.

Unlike the FSMs in figures 6.3 and 6.4, the FSM of figure 6.5 does not have the capability to make a 360° turn when there are no trail steps in sight. Like the FSM in figure 6.4, the only response of the FSM in figure 6.5 facing a marked trail step is to move onto it. When there are no marked trail steps around the ant, the state sequence 0-1-3-5-7 first turns the ant to the right, then follows with two left turns, and ends with a right turn, restoring the ant to its original position. This strategy (or parts of it) which tries to locate the next marked trail step is applied at steps g1, 52, g3, g4, g6, g9, g11, g16, g19, g20, g24, g25, g27, g29, g32, and g37. At the italicized steps, the subsequence 3-5-7 is invoked. Thus the ant looks only to the left for the missing trail before continuing on in its original direction. Sequences 0-2-3-6 and 1-4-3-6-2 along with the cycle 2-3-6 is used on several marked straightaways, but state 7 alone is also utilized to walk along some of the marked straightaways. The sequences 4-6-3 and 8-7-0 are used to move from a marked step across two successive unmarked steps. These are invoked at steps 79 (4-6-3), 80 (8-7-0), 81 (4-6-3), 82 (8-7-0), 83(4-6-3), 84 (8-7-0), 85 (8-7-0), 86 (4-6-3), 87 (4-6-3), and 88 (8-7-0). Right turns are navigated by either state 2 or 0 followed by state 1. The left turn at steps 32, g14, and g35 are handled by state 3 followed by state 5. The FSM of figure 6.5 does not have the ability to take an ant across three unmarked steps in a row, but this does not seem to hinder its performance. The FSMs, which were examined and

which have that capability, do not use it on the John Muir trail.

The FSMs in the next section contain only eight states. These are of interest because the memory-restricted storing of FSMs during the runs did not contain FSMs with fewer than eight states.

6.3 8-state perfect scoring FSMs

The smallest FSMs, which were seen during the examination of successful FSMs during the runs for the research, have eight states. At this point it is not known whether a smaller successful FSM exists for this problem.

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/1	2/2
1	3/1	4/2
2	5/2	1/2
3	5/0	2/2
4	5/2	6/2
5	7/2	7/2
6	0/0	1/2
7	0/0	6/2

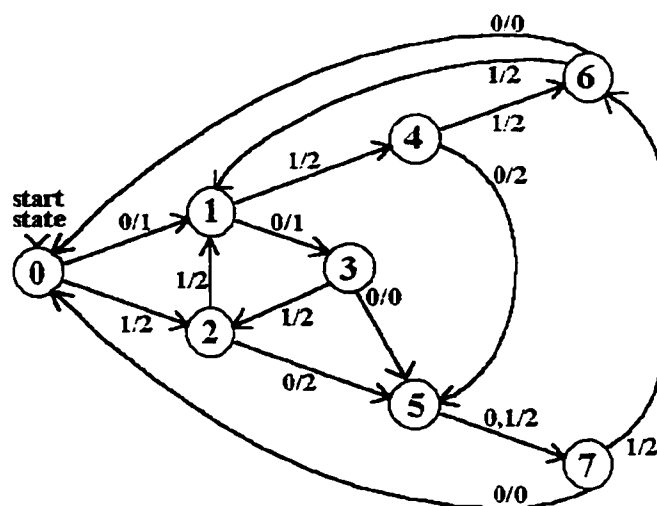


Figure 6.6: 8-state Perfect Scoring FSM with Time=193

The ant in figure 6.6 takes 193 time steps to traverse the trail. This ant, like the two previous ones, will always move onto a marked trail step immediately in front of it. This FSM does not contain any strategies which have not already been discussed. It is of interest because no 8-state machine was seen which needed less than 193 time steps to traverse the trail.

This FSM is of interest not only because it is one of a family of successful FSMs, but more so because parts of its family appeared in two very different runs. The two runs started with different initial populations. One run applied MTF to generation 0, and then only to every fourth generation after that. This run did not incorporate competition. The other run applied MTF to every generation and used competition during each generation to determine the fitness.

For all except two of the successful relatives, the only change in the state transition table is in the next state of state 5 with input 1. Both runs found the FSM in figure 6.6, and FSMs in which the next state for state 5 with input 1 is either 0 or 4. In addition, an FSM with this next state changed to 3 appeared in one of the runs. It might also have been bred by the other run and not been seen. Recall that only the first 10 successful FSMs from each generation were recorded, and not all were carefully scrutinized. One other variation which was seen replaces the next state for state 2 with input 1 from state 1 to 5. All of these ants required 193 time steps to traverse the trail. Other relatives of the FSM in figure 6.6 have changes in the next state, or next state and output for state 5 with input 1, but these relatives have times greater than 193. Changing the next state for state 5 with input 1 to state

Start state: 11

present state	next state/ output for input=0	next state/ output for input=1
4	12/2	17/2
11	23/1	20/2
12	21/2	25/2
17	11/0	23/2
20	12/2	23/2
21	11/0	17/2
23	25/1	4/2
25	12/0	20/2

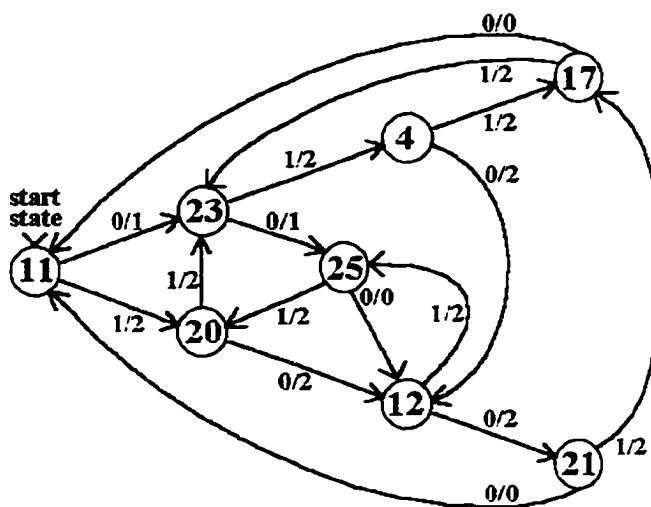


Figure 6.7: 8-state Perfect Scoring FSM with Time=193 Bred by a Benchmark Run

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/1	2/2
1	3/1	4/2
2	5/2	1/2
3	5/0	2/2
4	5/2	6/2
5	7/2	3/2
6	0/0	1/2
7	0/0	6/2

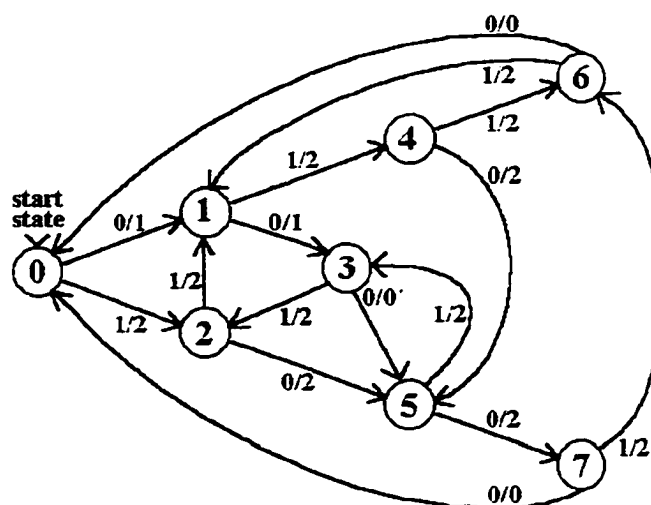


Figure 6.8: FSM of Figure 6.7 after MTF

5 yields an ant which has a time of 197. Replacing the next state with 6 and output with 3 for state 5 with input 1 results in an ant which takes the full allotment of 200 time steps to complete the trail.

A benchmark run produced the FSM in figure 6.7. Comparing the tables in figure 6.6 and 6.7 gives no hint of the similarity between the two machines, but the digraphs indicate that these FSMs are very similar. Applying MTF to the FSM in figure 6.7 yields the FSM in figure 6.8. Comparing figures 6.6 and 6.8 clearly indicates the similarity between these two FSMs produced by two different methods. As already indicated, the FSM in figure 6.8 was bred during an MTF run.

Note that the FSMs in figures 6.7 and 6.8 share only one state, state 4. The outputs and next states for state 4 are the only relevant bits they have in common. There are only eight bits in that part of the genome which have the same value—the four bits representing the outputs and four of the bits for the next states. The fact that these two machines are equivalent would do nothing for schema retention and development if they appeared in the same population.

The last FSM which is presented appears in figure 6.9. This FSM requires 199 time steps to cover the trail. As for some of the other FSMs presented, this FSM's only response to a marked trail step immediately ahead is to move onto it. This machine differs from all of those previously presented in that it does not make any left turns. Consequently a left turn can only be navigated with three right turns. The state sequence 5-7-0-1 (used at g1, g3, g14, g20, g25, g27, and g35), 2-0-1-3 (invoked at step 32) and 7-0-1-3 (not used) take an ant through a 270° right turn as long as the

input remains 0. Thus it takes three time steps to make a left turn. The only other point of interest is that the FSM of figure 6.9 uses a single state to walk along straightaways. For the first three straight trail sections the ant is in state 2. For other straightaways the ant is primarily in state 6.

Start state: 0

present state	next state/ output for input=0	next state/ output for input=1
0	1/0	2/2
1	3/0	4/2
2	0/0	2/2
3	5/2	5/2
4	6/2	6/2
5	7/0	1/2
6	5/2	6/2
7	0/0	4/2

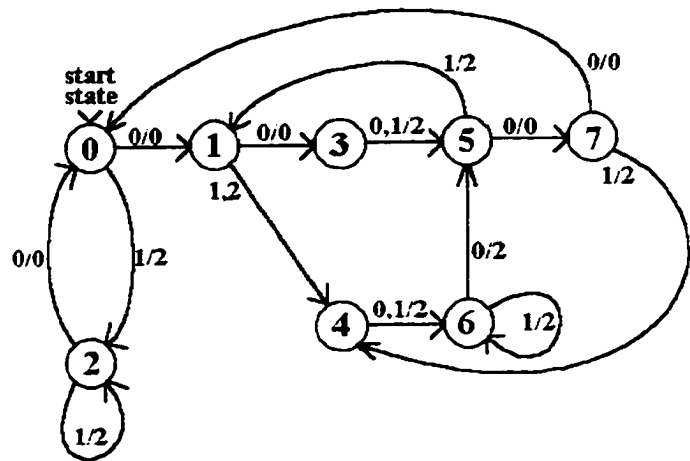


Figure 6.9: 8-state Perfect Scoring FSM with Time=199

The successful ants presented share some common strategies. One strategy which saves time is that anywhere along the trail after successfully navigating the corner at g3, an ant can move ahead onto two successive unmarked trail steps immediately following a marked trail step without losing the trail. Another shortcut strategy is to look in only one direction for the missing trail when the next trail step is not marked. These strategies are effective for the John Muir trail, but they are not

good general trail following strategies.

The FSMs of figures 4.4 and 4.5 always check to the left and right before moving ahead when the trail step immediately ahead is not marked. As previously mentioned, this consumes a great deal of time since it takes five time steps to move to an unmarked step directly ahead. Looking to the left and right to see if the trail turned when the next step is not marked is a better strategy for all trails, but it creates a problem when there is a time limit.

This chapter indicates that GAs locate strategies to solve a problem in the process of finding a solution to the problem. Running a GA with different initial populations can produce different solutions implementing similar strategies. While this is an interesting point it is not the main purpose of this research. The main goal of the research is to determine the effect of the changes made to the benchmark GA (chapters 4 and 5) on the GA's efficiency. The next chapter describes the criteria used to evaluate the the new algorithms.

>>>●<<<

Chapter 7

Evaluation criteria

The main purpose of this thesis is to determine whether the new reorganization operators (SFS and MTF, chapter 5), possibly combined with competition, can positively affect the efficiency of a GA utilizing the FSM genome. Different variations and combinations of SFS, MTF, and competition were incorporated into a benchmark GA which was based on Jefferson et al. (1992). The numerous methods which resulted and which are to be compared necessitate the development of criteria to measure the relative performance of these methods in terms of efficiency.

This chapter concentrates on the criteria used to evaluate the relative performance/efficiency of the different strategies. Efficiency is measured in terms of the number of generations needed to locate a solution and/or the amount of processor time used to get the solution. Augmenting the benchmark with a reorganization operator (MTF or SFS, chapter 5) and/or competition adds additional processing to the benchmark. Gains made by SFS or MTF in terms of the number of generations to a solution might not translate into less processor time to a solution. Fewer generations combined with less processor time is considered the most efficient. In addition the ability to locate a solution with fewer states is considered an

advantage.

The next section (7.1) gives a brief description of the experiments and of the type of data collected to give a foundation for the presentation of the criteria used to evaluate the data (sections 7.2 and 7.3).

7.1 The context

To determine the effect of the different methods on the GA, different types of experiments were conducted, and the results are evaluated in different ways. This chapter describes the evaluation criteria and how they will be used to determine the relative effectiveness of the different methods (sections 7.2 and 7.3). This section presents the type of data to which the criteria will be applied. A more detailed description of the experiments, and the evaluation of the results appear in chapter 8.

Two different types of experiments were designed to collect two different types of data for two different purposes. First, data were collected for each generation for a fixed number of generations to track the evolutionary process. (Samples of the output are shown in figure 4.25 in section 4.3, and in figures 5.8 and 5.9 in section 5.3.) Second, data were then collected with respect to only the outcome of each run in order to monitor successful and unsuccessful completions. For this second type of experiment the GA was terminated either when a solution was found or when the preset loop limit for the number of generations was reached. Figure 7.1 contains a sample printout for a 30-seed run for this second type of

experiment.

Benchmark-timed
 Statistics on file blqt30.st
 Crossover rate=0.01; Mutation rate=0.01
 Population size=16384
 Select the top 820 individuals or 5.00% of each generation for reproduction.

Last generation = 125

Run#	Seed	Run Time	#PS/in	gen#/min	#states	Run time 100 gen
1	0.74903538	2113.660000	0			1691.87
2	0.39438263	1776.202704	1/	105 /19		1691.85
3	0.69343845	780.960000	1/	46 /26		780.96
4	0.03923732	2113.822704	0			1692.46
5	0.57838453	2117.990000	0			1695.30
6	0.29652832	2112.682704	0			1691.62
7	0.48296342	2115.160000	0			1692.87
8	0.13742938	1541.850000	1/	91 /23		1541.85
9	0.80347386	1471.302704	1/	87 /28		1471.30
10	0.93847534	2115.900000	0			1693.49
11	0.61563214	1118.952704	1/	66 /24		1118.95
12	0.27384864	1473.020000	1/	87 /25		1473.02
13	0.37929661	2116.080000	0			1693.68
14	0.41974869	1996.322704	1/	118 /21		1692.33
15	0.20203185	2115.910000	0			1692.93
16	0.72399187	2112.522704	0			1690.53
17	0.32661661	1911.010000	1/	113 /13		1691.67
18	0.55001424	2118.422704	0			1695.22
19	0.19171490	2113.740000	0			1691.62
20	0.82574681	1574.142704	1/	93 /14		1574.14
21	0.65795069	1557.230000	1/	92 /24		1557.23
22	0.00132096	2115.360000	0			1693.31
23	0.44038342	1438.282704	1/	85 / 9		1438.28
24	0.09711350	2114.460000	0			1692.33
25	0.78874573	2118.302704	0			1695.31
26	0.51412962	2116.550000	0			1694.19
27	0.90991612	1626.282704	1/	96 /26		1626.28
28	0.16076033	797.840000	1/	47 /23		797.84
29	0.99843391	2115.560000	0			1693.93
30	0.85823880	2117.722704	0			1694.11

For last generation = 125

number of successful runs = 13

% of successful runs = 43.33

number of failed runs = 17

% of failed runs = 56.67

g_av = 86.62 generations, standard dev = 21.20

t_av_s = 1466.42 seconds, standard dev = 357.77

t_av_f = 2115.52 seconds, standard dev = 1.82

avg machine size = 21.153846, standard dev = 5.558724

For last generation = 100

number of successful runs = 10

% of successful runs = 33.33

Figure 7.1: Example of a 30-Seed Run (continued on next page)

```

number of failed runs = 20
% of failed runs = 66.67
g_av = 79.00 generatioins, standard dev = 18.01
t_av_s = 1337.99 seconds, standard dev = 304.07
t_av_f = 1693.03 seconds, standard dev = 1.33
avg machine size = 22.200000, standard dev = 5.653318

```

Figure 7.1 continued: Example of a 30-Seed Run

In figure 7.1, **#PS** is the number of solutions which were found. **min #states** is the smallest successful FSM which appeared; for example if five successful FSMs appeared, the size of the smallest one was recorded. Data is available for different values of *maxgen*; *maxgen* is the last generation the GA is permitted to breed before the iterative process is automatically halted. This was included to measure the sensitivity of the efficiency of the different methods to *maxgen*. The **Run Time** in the third column is for the value of *maxgen* printed as **Last generation**. The **Run Time** in the last column of the printout is for the smaller value of *maxgen*.

In the summary which appears after the per seed printout, **t_av_f** is the average processor time needed for a run which did not find a solution by generation *maxgen*, **g_av** is the average of the numbers which appear in the column labelled **in gen#** and which are $\leq \text{maxgen}$, and **t_av_s** is the average processor time corresponding to **g_av**. **avg machine size** is the average of the numbers which appear in the column labelled **min #states** and which correspond to the values used to calculate **g_av**.

In the next section (7.2), the criteria for evaluating the evolutionary process

is discussed. In the following section (7.3), the much more complex task of evaluating efficiency is developed.

7.2 Evaluating the evolutionary process

As described in section 7.1, data were collected for each generation to permit a comparison of how the different methods affect the evolutionary process as the GA breeds generation after generation. To this end, two sets of experiments were carried out. Each experiment consisted of 30 runs per method. One experiment was executed with a population of 64K FSMs; for this population size, 70 generations were bred for each of the 30 runs. The other experiment had a population of 16K individuals and bred 125 generations after generation 0. These experiments are further detailed in section 8.1.1. The corresponding data and the analysis of these data appear in appendix C and section 8.1.2, respectively.

The evolutionary process is viewed on graphs. Two different types are used. One type consists of a set of 30 graphs with each containing the data for some set of individual runs. (See appendix C.) For the other type of graph the maximum fitness appearing in each generation is averaged over the 30 runs for a given method. (See section 8.1.2.)

The first type of graph permits comparison of individual runs: three sets are provided for each experiment. The runs for each graph are chosen based on 1) run number, 2) evolutionary progress, or 3) location of optimal individual. The reader is referred to appendix C for the discussion in this section. The data on these graphs

are evaluated in section 8.1.2.

The graphs (appearing in figures C.1a, $i \in \{1,2,3,4,5,6\}$, in appendix C), which contain a single run number across the methods, actually compare the runs which have the same initial population. This is accomplished by using the same seed for the random number generator across all runs with the same run number and same population size.

The second type of graph containing individual runs compares the runs based on evolutionary progress. For each run, the highest fitness which appears to have started to move the evolutionary process ahead is recorded, along with the first generation in which it started the process of increasing its representation in the population. This fitness is not necessarily the maximum fitness to appear. If the maximum fitness which is bred appears only once, or intermittently (See example in next paragraph.), it is not recorded. Similarly, the generation recorded corresponding to the recorded fitness is not necessarily the first generation in which that fitness appeared.

For example, consider the experiment in figure 5.9 (section 5.3). The recorded fitness is 86 appearing in generation 39. The 86 was chosen because it was the last fitness that appeared to dominate the evolutionary process. The fitness of 87 appears intermittently, but it does not seem to move the evolutionary process along. Consequently 87, the maximum fitness bred, is ignored for this part of the evaluation. If this run had been permitted to execute for additional generations and the fitness of 87 appeared in increasing amounts from generation 69 on, the recorded

fitness would be 87 with corresponding generation 69. The fact that 87 appears before generation 69 would be ignored since it does not maintain a constant and increasing presence in the population prior to generation 69.

Once the data are recorded as described in the previous two paragraphs, the runs for each method are sorted in decreasing order by the recorded fitness, and within each fitness in increasing order by the corresponding generation. Then for the set of graphs comparing individual runs based on evolutionary progress, graph j contains the j^{th} run on the sorted list of each of the methods being compared. These graphs appear in figures C.*ib*, $i \in \{1,2,3,4,5,6\}$ in appendix C.

The third set of graphs comparing individual runs (appearing in figures C.*ic*, $i \in \{1,2,3,4,5,6\}$, in appendix C) are based on the location of an optimal individual. The first generation in which a perfect scoring individual appears is recorded, along with the highest fitness which appears to move the evolutionary process ahead and the generation in which this is initiated (as described in the previous paragraphs). These runs are separated into two groups: those runs which found a solution and those which did not. Those runs which found a perfect scoring individual are sorted by increasing generation in which the solution was found. Those runs which did not find a solution are sorted as described in the previous paragraph. A single list is formed by appending the sorted list of unsuccessful runs to the end of the sorted list of successful runs. Then, as for the previous set of graphs, the j^{th} graph contains the j^{th} entry on each list of the methods being compared (section 8.12).

Another type of graph is also generated similar to a method used by Levinson

(1994). For this second set of graphs, the maximum fitness attained in each generation is averaged over the 30 runs for a method. The averages for each generation are plotted for the methods being compared.

For both types of graphs described in this section, a limited number of methods are compared at a time in order to produce readable data. One set of graphs compares the benchmark, SFS, and MTF—all three without competition. A second set compares the benchmark and SFS, both with and without competition. A third set compares the benchmark and MTF, both with and without competition. In addition, the MTF algorithm is applied at different generation intervals (every generation, every other generation, through every fifth generation). The averages over the 30 runs for each of the MTF variants are also compared.

The evaluation of the evolutionary process appears in section 8.12, and is based on the data presented in appendix C (comparison of individual runs) and in section 8.12 (averages over 30 runs). Tracking the evolutionary process provides some insight to the relative performance of the different methods, but the real concern is whether one method generally causes the GA to converge in fewer generations, and also whether one method generally uses less processor time to find a solution. The next section formalizes the evaluation concerning the convergence of the GA.

7.3 Evaluating convergence

The evaluation of the evolutionary process as described in the previous section

is valuable, but the effect of the different methods on efficiency/convergence is the major concern. Evaluating the effect of each of the methods on the rate of convergence is rather complex because not all runs successfully find a solution. Different approaches were considered before deciding how to evaluate the convergence data in this thesis. These approaches deserve mention in their own right since they can possibly be applied to other situations of a genetic algorithm application. The next subsection (7.3.1) presents some of the variables necessary for the evaluations that will be described in the subsequent subsections.

7.3.1 Evaluation factors

There are several different averages which can be considered for the purpose of evaluating the efficiency of the different methods. These averages are introduced in this section. Traditional statistical analysis forms the basis for the formulas which are developed (sections 7.3.5-7.3.9) for the purpose of evaluation of the data. Populations of 64K, 32K, and 16K individuals were created using two different sets of seeds: the original 30 seeds used for tracking the evolutionary process, and 100 seeds which did not include any of the 30 previously used seeds.

Several factors must be considered to evaluate convergence. There was a concern as to how dependent the ranking of the different methods would be on *maxgen*, the maximum number of generations that the GA is permitted to breed. Consequently, calculations are computed and saved for *maxgens* of 50, 60, and 70 for a 64K population, for *maxgens* of 75 and 100 for a population of 32K FSMs, and for

a 16K population the *maxgens* are 100 and 125. In addition the following factors are considered:

- 1) For runs that found a solution within *maxgen* generations,
 - a) G_{AV} is the average generation in which a solution is found; this is calculated by taking the average over those runs which found a solution.
 - b) $T_{AV:S}$ is the corresponding processor times averaged over the same runs.
- 2) To take into consideration the fact that not all seeds will produce a run which finds a solution within *maxgen* generations,
 - a) $G\mu$ is the expected mean number of generations required before a solution is found; this involves using an additional seed each time a run fails to find a solution.
 - b) $T\mu$ is the corresponding processor time.
- 3) In addition, MTF, which moves relevant data to the front of the genome, should have shorter schemata and thus is expected to encourage smaller machines. Therefore the number of states for successful FSMs is also tracked.

The calculations for G_{AV} , $T_{AV:S}$, $T_{AV:F}$ (average time for runs which did not find a solution), and average machine size, and their variances use the formulas:

$$\text{average} = x_{AV} = \frac{\sum_{i=1}^n x_i}{n} \quad \text{and} \quad \text{variance} = \frac{\sum_{i=1}^n (x_i - x_{AV})^2}{n}.$$

The calculation of $G\mu$, $T\mu$, and their variances is more complex since it is not known *a priori* how many failed runs will precede a successful run. This issue is addressed next.

7.3.2 Considerations in the calculation of $G\mu$ and $T\mu$

When all runs successfully find solutions, averages and variances are easy to calculate using the formulas presented in the previous paragraph. But when one or more runs fail to find a solution, calculating these measures for average performance are much more complex. In order to find formulas for these measures the course of experimentation must be clearly defined.

Assume that a researcher, trying to improve GA efficiency, studies a GA which will halt either when a solution to a problem is found, or when some preset generation, generation *maxgen*, has been evaluated by the GA. Such research employs the following course of experimentation:

- 1) Execute the genetic algorithm for an initial method (with fixed parameter values) for n initial populations.
- 2) Change some earlier tested method of the genetic algorithm (retaining the same parameter values) and execute it for n initial (possibly new) populations.
- 3) Repeat step 2 for each variation under consideration.
- 4) In the event that all n runs for a given method of an experiment fail to find a solution for that method, either discard the method as being nonviable under the given conditions, or change one or more parameters and rerun all experiments starting from step 1 using the new parameters.

This uniform application of the GA to different methods will provide the data for the individual methods which forms the base for the comparison of the methods. The focus now moves to the calculations for the individual methods and for pairs of

methods which will provide the actual values used for the comparisons. As a result of this course of experimentation each method will have an associated failure rate, that is the number of runs which failed to find a solution. When there are different failure rates for the different methods, the mean number of generations and/or amount of processor time needed for a given method to find a solution will be highly dependent on the failure rate.

Since some of the n runs could execute through generation *maxgen* and be terminated without finding a solution, the mean number of generations (mean amount of processor time) needed to find a solution should combine the number of generations (amount of processor time) due to failed runs and the number of generations (amount of processor time) for a successful run following the failed runs. Within this context the calculation of the mean ($G\mu$, $T\mu$) and variance for the population of all seeds requires a summation involving an infinite number of terms. In order to find closed forms for $G\mu$ and $T\mu$ and their variances for each set of n runs, closed forms for the sums of specific infinite sequences are needed. In the next section (7.3.3) the closed forms for these sums are presented.

7.3.3 Formulas for the summation of the required infinite sequences

To develop the formulas for $G\mu$ and $T\mu$ (defined in 7.3.1) and their variances the summation of three infinite sequences of probabilities are needed. The problem-specific probabilities are substituted into these formulas in section 7.3.5.

The first sum which is needed, $\sum_{i=0}^{\infty} r^i$, is the sum of an infinite geometric

progression with a first term of 1 and ratio of r , $0 < r < 1$. Even though the formula for this sum is well known, it is included for completeness.

Equation 7.1:
$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}, \quad 0 < r < 1.$$

The two additional formulas which are needed to find $G\mu$, $T\mu$, and their variances are

Equation 7.2:
$$\sum_{i=1}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}, \quad 0 < r < 1, \text{ and}$$

Equation 7.3:
$$\sum_{i=1}^{\infty} i^2 r^{i-1} = \frac{r+1}{(1-r)^3}, \quad 0 < r < 1.$$

The proofs for these equations are presented in appendix B.

The formulas for the desired means and variances can now be developed by utilizing equations 7.1, 7.2, and 7.3. In the next section (7.3.4) the variables, which are needed for the development of these formulas, are defined within the context of the experimentation.

7.3.4 Definitions of variables

Before developing formulas (sections 7.3.5 through 7.3.7, and 7.3.9) for the desired means ($G\mu$ and $T\mu$ as defined in section 7.3.1) and their variances, the variables used in these formulas must be defined. These variables are described in this section.

In the course of the experimentation described in 7.3.2, the GA will halt after processing generation g_i if a solution is found in generation g_i . Otherwise it will halt after processing generation *maxgen*, the preset loop limit. Let F be the number of

runs which fail to find a solution. These F runs will terminate immediately after generation *maxgen* has been processed. Since there are n runs in total, $S=n-F$ runs successfully find a solution during or before generation *maxgen*.

Assuming a properly distributed random sample forming the n runs, the probability that a run will execute through generation *maxgen* due to failure can be estimated using $p_F = \frac{F}{n}$, and the probability that a run successfully finds a solution to a problem is estimated as $p_S = \frac{S}{n} = 1 - p_F$. When $F=n$, $S=0$. If this occurs, the GA failed to find a solution during any of the n runs; therefore the method failed and further analysis is not necessary.

When $F=0$, $S=n$. In this case, all n runs found a solution and the formulas (given in section 7.3.1) for the average and variance can be applied using the n values which are available. The formulas which need to be developed for the mean and variance assume $0 < S, F < n$ and $0 < p_S, p_F < 1$, a restriction required by the limitations on r in equations 7.1 through 7.3 (section 7.3.3). Hence, the case of $F=0$ will be treated separately in the proofs of these formulas in order to develop full formula definition.

Let $G_{AV} = \frac{\sum_{i=1}^S g_i}{S}$ which is the average generation in which a solution is found. As indicated this average is taken over the S successful runs. A genetic algorithm is started with generation 0, therefore the average number of generations processed by a run which successfully finds a solution is $G_{AV}+1$, and a run which fails to find a solution processes *maxgen*+1 generations. The GA will generate on the average $G\mu$ generations in order to find a solution. The average G_{AV} only

considers successful runs, but $G\mu$ is necessary because some of the n test runs failed to find a solution.

The development of the formulas for $G\mu$ and its variance assumes that a researcher who is trying to find a solution to a problem first writes a problem-specific GA. The GA is then executed. If the GA completes generation *maxgen* without finding a solution, it is rerun using a different initial population. This is repeated until a solution is found in some generation less than or equal to *maxgen*. For each repetition, only the initial population is changed. In general applications when a run fails to find a solution, a researcher frequently changes more than the initial population for the next run; *maxgen*, population size, mutation rate, or the selection process are prime candidates for alteration. This thesis assumes that no changes will be made except the initial population in order to develop the formulas to compare the different methods.

In the next three sections the formulas for $G\mu$ (section 7.3.5), $T\mu$ (section 7.3.7), and their variances (sections 7.3.6 and 7.3.7) are developed.

7.3.5 Calculating $G\mu$, the mean number of generations

To find formulas for $G\mu$ and its variance, equations 7.1 through 7.3 must be brought into the context of the genetic algorithm. To this end let $r=p_F$ =the probability that a run fails to find a solution by generation *maxgen* (the preset loop limit), and $1-r=p_S$ =the probability that a run finds a solution in generation $g \leq \text{maxgen}$. Making these substitutions into equations 7.1 through 7.3 yields the

following three equations respectively:

Equation 7.4: $\sum_{i=0}^{\infty} p_F^i = \frac{1}{p_S}, 0 < p_F, p_S < 1,$

Equation 7.5: $\sum_{i=1}^{\infty} i p_F^{i-1} = \frac{1}{p_S^2}, 0 < p_F, p_S < 1. \text{ and}$

Equation 7.6: $\sum_{i=1}^{\infty} i^2 p_F^{i-1} = \frac{p_F + 1}{p_S^3}, 0 < p_F, p_S < 1.$

As explained earlier, the limitations on p_F and p_S are realistic. If $p_F=0$ and $p_S=1$, all test runs successfully found a solution, and the average and variance can be calculated using the formulas involving finite sums. If $p_F=1$ and $p_S=0$, all test runs failed to find a solution by generation *maxgen*. In this case it is rather unlikely that the tested GA will find a solution without additional modifications, for example to *maxgen*, the population size, the selection process, or the mutation rate.

Theorem: The mean number of generations needed to find a solution if a GA is permitted to process a maximum of *maxgen*+1 generations is

$$G\mu = G_{AV} + 1 + \frac{F}{S} (\text{maxgen} + 1).$$

Proof: $G\mu$ is the mean number of generations that a genetic algorithm is expected to generate in order to find a solution to a problem. To calculate this use the formula $\text{mean} = \sum_i p_i x_i$, where p_i is the probability associated with the value x_i (Freund 1962). Then

$$G\mu = \sum_{i=0}^{\infty} p_i \times (\text{number of generations processed by a successful run preceded by } i \text{ unsuccessful runs}).$$

On the average a successful run will process $G_{AV} + 1$ generations. Therefore

$$G\mu = \sum_{i=0}^{\infty} p_F^i p_S [G_{AV} + 1 + i(\text{maxgen} + 1)]$$

$$= p_S(G_{AV}+1) \sum_{i=0}^{\infty} p_F^i + \sum_{i=0}^{\infty} p_F^i p_S[i(maxgen+1)].$$

By equation 7.4 $\sum_{i=0}^{\infty} p_F^i = \frac{1}{p_S}$. Therefore the first term of G_{μ} reduces to $G_{AV}+1$.

Thus, G_{μ} can be viewed as the sum of the average of two random variables: 1) random variable G is the number of generations for a successful run, and 2) random variable I is the number of generations for the unsuccessful runs preceding a successful run. (The fact that G_{μ} is the sum of the averages of two random variables is critical in the development of the formula for the variances in sections 7.3.6 and 7.3.9. Hence the proof will be developed from this point of view.) Thus,

$$\begin{aligned} G_{\mu} &= \text{average number of generations processed by a successful run} \\ &\quad + \text{average number of generations processed by failed runs} \\ &\quad \text{preceding the successful run} \\ &= G_{AV}+1 \\ &\quad + \Sigma (\text{probability that a given number of generations will be processed} \\ &\quad \quad \text{by failed runs preceding the successful run} \times \\ &\quad \quad \text{that number of generations}) \\ &= G_{AV}+1 \\ &\quad + \text{probability of failure on the first run} \times \\ &\quad \quad \text{probability of success on the second run} \times \\ &\quad \quad \text{the number of generations processed by the single unsuccessful run} \\ &\quad + \text{probability of failure on the first two runs} \times \\ &\quad \quad \text{probability of success on the third run} \times \\ &\quad \quad \text{the number of generations processed by the two unsuccessful runs} \\ &\quad + \dots \\ &\quad + \text{probability of failure on the first } i \text{ runs} \times \\ &\quad \quad \text{probability of success on the } i+1^{\text{st}} \text{ run} \times \\ &\quad \quad \text{the number of generations processed by the } i \text{ unsuccessful runs} \\ &\quad + \dots \\ &= G_{AV}+1 \\ &\quad + p_F p_S (maxgen+1) + p_F^2 p_S [2(maxgen+1)] + p_F^3 p_S [3(maxgen+1)] \\ &\quad + \dots + p_F^i p_S [i(maxgen+1)] + \dots \end{aligned}$$

$$\begin{aligned}
&= G_{AV}+1 + p_F p_S(maxgen+1)(1+2p_F+3p_F^2+\dots) \\
&= G_{AV}+1 + p_F p_S(maxgen+1) \sum_{i=1}^{\infty} i p_F^{i-1}.
\end{aligned}$$

Applying equation 7.5 yields

$$G_{\mu} = G_{AV}+1 + p_F p_S(maxgen+1) \left(\frac{1}{p_S^2} \right) = G_{AV}+1 + \frac{p_F}{p_S}(maxgen+1).$$

Since $\frac{p_F}{p_S} = \frac{F}{S}$, the final formula for the mean is $G_{\mu} = G_{AV}+1 + \frac{F}{S}(maxgen+1)$.

In the best situation, all test runs successfully find a solution yielding $G_{\mu} = G_{AV}+1$, but the proof of this formula assumes that $F \neq 0$ since equation 7.5 assumes $F \neq 0$. Since it is reasonable to define $G_{\mu} = G_{AV}+1$ when $F=0$ (as a special case), the formula is then also valid for $F=0$. $\square$

In addition to G_{μ} , statistical analysis requires the corresponding variance in order to employ traditional methods. The formula for the variance is developed in the next section.

7.3.6 Calculating the variance

The formula for the variance is developed analogous to the manner in which the formula for G_{μ} of the previous section (7.3.5) was derived. The formula for the variance of a single random variable is $\left(\sum_i p_i x_i^2 \right) - x_{\mu}^2$, where $x_{\mu} = \sum_i p_i x_i$ and p_i is the probability associated with x_i (Freund 1962). But it is shown at the beginning of the proof of the formula for G_{μ} (section 7.3.5) that G_{μ} is the sum of the averages for two random variables: 1) random variable G is the number of generations for a successful run, and 2) random variable I is the number of

generations for the situation that unsuccessful runs preceded a successful run. Therefore it is necessary to start with the formula for the variance for this situation.

Consider the mean, $X_{\mu} = X_{AV:1} + X_{AV:2}$. X_{μ} is the sum of the means of two random variables. In this case $\text{var}(X_{\mu}) = \text{var}(X_{AV:1}) + 2\text{cov}(X_{AV:1}, X_{AV:2}) + \text{var}(X_{AV:2})$. When the two events which are used to calculate $X_{AV:1}$ and $X_{AV:2}$ are independent, their covariance is zero (Freund 1962). Fortunately this is the case for G_{μ} .

Execution of a GA involves randomly choosing a seed from an infinite set of seeds. When a finite number of randomly selected elements is removed from an infinite set, the probabilities associated with properties of the set are not altered. Thus the probability that the $i+1^{\text{st}}$ seed randomly chosen to start a GA locates a solution in generation g , given that the i previously selected seeds produced unsuccessful runs, is the same as the probability that the $i+1^{\text{st}}$ randomly chosen seed locates a solution in generation g . Similarly the probability that the $i+1^{\text{st}}$ randomly chosen seed fails to locate a solution, given that the i previously selected seeds produced unsuccessful runs, is the same as the probability that the $i+1^{\text{st}}$ randomly chosen seed fails to locate a solution. When the probability that event E occurs given that event F occurred is the same as the probability that event E occurs, events E and F are independent. Therefore the events G (a specific number of generations for a successful run) and I (a specific number of generations for unsuccessful runs preceding a successful run) used in the calculation of G_{μ} are independent. Consequently the variance of G_{μ} reduces to the sum of the variance for the average

number of generations for the successful runs and the variance for the number of generations expected for situation that the unsuccessful runs preceding a successful run: that is $\text{var}(G\mu) = \text{var}(G) + \text{var}(I)$.

To find the variance for $G\mu$, a closed form for $\sum_i p_i x_i^2$ for the unsuccessful runs is needed in terms of the GA variables. Within the context of the genetic algorithm, x_i is the number of generations which the GA will process for unsuccessful runs. Once the following lemma is verified, the formula for $\text{var}(G\mu)$ can be proven.

Lemma: When randomly selecting seeds for a GA to locate a solution to a problem, for the $i = \{0, 1, 2, 3, \dots\}$ unsuccessful runs which can precede a successful run, the term $\sum_i p_i x_i^2$ needed for the calculation of the variance of a single random variable

$$= \frac{F(F+n)}{S^2} (\text{maxgen}+1)^2, F \neq 0.$$

Proof: $\sum_i p_i x_i^2 =$

$$\sum [\text{probability that a given number of generations will be processed before a successful run} \times (\text{that number of generations})^2]$$

$$\begin{aligned} &= \text{probability of failure on the first run} \times \\ &\quad \text{probability of success on the second run} \times \\ &\quad (\text{the number of generations processed by the unsuccessful run})^2 \\ &+ \text{probability of failure on the first two runs} \times \\ &\quad \text{probability of success on the third run} \times \\ &\quad (\text{the number of generations processed by the 2 unsuccessful runs})^2 \\ &+ \dots \\ &+ \text{probability of failure on the first } i \text{ runs} \times \\ &\quad \text{probability of success on the } (i+1)^{\text{st}} \text{ run} \times \\ &\quad (\text{the number of generations processed by the } i \text{ unsuccessful runs})^2 \\ &+ \dots \end{aligned}$$

$$\begin{aligned} &= p_F p_S [\text{maxgen}+1]^2 + p_F^2 p_S [2(\text{maxgen}+1)]^2 \\ &+ p_F^3 p_S [3(\text{maxgen}+1)]^2 + \dots + p_F^i p_S [i(\text{maxgen}+1)]^2 + \dots \end{aligned}$$

$$= p_F p_S (\maxgen + 1)^2 (1 + 2^2 p_F + 3^2 p_F^2 + 4^2 p_F^3 + \dots + i^2 p_F^{i-1} + \dots)$$

$$= p_F p_S (\maxgen + 1)^2 \sum_{i=1}^{\infty} i^2 p_F^{i-1}.$$

Applying equation 7.6 yields $p_F p_S (\maxgen + 1)^2 \left(\frac{p_F + 1}{p_S^3} \right)$. Substituting $\frac{F}{n}$ for p_F .

and $\frac{S}{n}$ for p_S , and reducing to lowest terms results in the final form for $\sum_i p_i x_i^2$

which is $\frac{F(F+n)}{S^2} (\maxgen + 1)^2$. $\square$

Theorem: The variance for $G\mu$ is $\text{var}(G_{AV} + 1) + \frac{F n}{S^2} (\maxgen + 1)^2$.

Proof: The variance for $G\mu$ is the sum of the variance for the successful runs and the variance for the unsuccessful runs preceding a successful run. The general formula for the variance for a single random variable is $\left(\sum_i p_i x_i^2 \right) - x_\mu^2$ (Freund 1962), where in the context of the GA for unsuccessful runs $x_\mu = \frac{F}{S} (\maxgen + 1)$ (from the theorem in section 7.3.5), and $\sum_i p_i x_i^2$ is $\frac{F(F+n)}{S^2} (\maxgen + 1)^2$ (from the previous lemma). Making these substitutions yields

$$\begin{aligned} \text{var}(G\mu) &= \text{var}(G_{AV} + 1) + \frac{F(F+n)}{S^2} (\maxgen + 1)^2 - \left(\frac{F}{S} (\maxgen + 1) \right)^2 \\ &= \text{var}(G_{AV} + 1) + \frac{F n}{S^2} (\maxgen + 1)^2. \end{aligned}$$

In the best situation, all test runs successfully find a solution yielding $\text{var}(G\mu) = \text{var}(G_{AV} + 1)$, but the formula assumes that $F \neq 0$ since equation 7.6 assumes $F \neq 0$. Since it is reasonable to define $\text{var}(G\mu) = \text{var}(G_{AV} + 1)$ when $F = 0$ (as a special case), the formula is then also valid for $F = 0$. $\square$

The formulas for $G\mu$ (section 7.3.5) and its variance provide a way to apply standard statistical tests to determine the relative efficiency of different strategies

where efficiency is defined as finding a solution in the fewest number of generations. But strategies which are most efficient by this standard may be doing a great deal of additional work, and hence require more processor time to do the job. Therefore efficiency should also be evaluated in terms of processor time. This issue is discussed in the next section.

7.3.7 Calculation of mean processing time and its variance

$G\mu$ (section 7.3.5), the average number of generations needed to find a solution, is one way to gage efficiency, but the processor time corresponding to $G\mu$ is also an important measure of efficiency. In this section formulas are presented, which can be used to calculate both the mean amount of processor time needed to find a solution and the corresponding variance.

For a given method with n test runs, $n=S+F$, where S is the number of successful runs and F is the number of runs which failed to find a solution. Let the S successful runs have times t_i and the remaining F runs have times t_j . Define the average processor time for a successful run as $T_{AV:S} = \frac{\sum_{i=1}^S t_i}{S}$ and the average time for a run which does not find a solution by generation *maxgen* as $T_{AV:F} = \frac{\sum_{j=1}^F t_j}{F}$.

Theorem: Let $T\mu$ = the mean processor time needed to find a solution when multiple runs are needed.

$$1) T\mu = T_{AV:S} + \frac{F}{S} T_{AV:F}.$$

$$2) \text{ The variance for } T\mu \text{ is } \text{var}(T_{AV:S}) + \frac{Fn}{S^2} (T_{AV:F})^2.$$

Proofs: Replacing $G_{AV}+1$ and *maxgen*+1 with $T_{AV:S}$ and $T_{AV:F}$ respectively in the

proofs of the formulas for $G\mu$ (section 7.3.5) and its variance (section 7.3.6) yields the proofs for these two results. $\square$

The formulas derived for $G\mu$ (section 7.3.5) and its variance (section 7.3.6) are based on the fact that *maxgen* is a constant. However $T_{AV:F}$, which replaces *maxgen*, is not a constant due primarily to system resources available at any given moment. Because the standard deviation for $T_{AV:F}$ is very small (See figure 7.1 of section 7.1 for sample values.), it is reasonable to treat $T_{AV:F}$ as a constant in order to greatly simplify the formulas, and hence the calculations.

This mean and variance for processor time provide yet another measure of GA efficiency. In the next section (7.3.8) the reasons for not using the formulas for $G\mu$, $T\mu$, and their variances to evaluate the experiments are discussed.

7.3.8 Use of $G\mu$ and $T\mu$

The formulas for $G\mu$ and $T\mu$ and their variances, which were developed in the previous three sections (7.3.5 through 7.3.7), are not practical for the experiments which were carried out. The purpose of finding these values is to be able to apply standard statistical tests to compare the efficiency of the different methods (benchmark, SFS, MTF). For these tests to give some reasonable degree of certainty the variances must be comparatively small. Unfortunately these formulas yield very large variances. In addition the distribution due to the failed runs is a geometric distribution (due to the term $\sum_{i=1}^{\infty} i p_F^{i-1}$ appearing in the proofs in previous sections). Lacking a normal distribution and small variances, standard statistical tests using

these calculated values are not practical.

While the population of seeds is not normally distributed, a normal distribution can be generated. Mann's (1995) description of the Central Limit Theorem states that "the sampling distribution of the sample mean $[x_{AV}]$ is approximately normal, irrespective of the shape of the population distribution" (p. 378). Let $x_{AV;i}$ be the averages for a set of samples from some population. Then for a set of samples with $n \geq 30$ elements per sample, "the shape of the distribution of $[x_{AV;i}]$ is approximately normal irrespective of the shape of the population distribution" (Mann 1995, p. 379). Thus for many samples, the averages of these samples will form an approximately normal distribution. In addition by increasing n , the variance for this distribution can be reduced. Hence to be able to have a normal distribution with small variances so that traditional statistical tests can be used, the formulas for $G\mu$, $T\mu$, and their variances must be applied to a set of sample averages as opposed to a set of members of a single sample. To get small variances, a large number of samples in which each sample consists of a large number of seeds would have to be used. This is computationally very intensive and was avoided for this study.

The formulas for $G\mu$ and $T\mu$ and their variances (sections 7.35 through 7.37) assume that the single small sample used for the calculations is representative of the population of all seeds. The different values for the failure rate obtained from the experimentation with different sets of seeds indicate that this assumption cannot readily be made. In an attempt to adapt the theory of sections 7.3.5 through 7.3.7

to the data collected for this thesis, the next section (7.3.9) develops similar formulas to calculate these statistical values for an individual sample.

7.3.9 $G\mu$ and $T\mu$ formulas for a sample

Since it is not feasible to use the formulas for $G\mu$ and $T\mu$ (as developed in sections 7.3.5 through 7.3.7) which make assumptions about the whole population of seeds based on a small sample, the next step is to focus on the sample. The formulas for $G\mu$ and $T\mu$ and their variances which are developed in this section for a sample are different from those of the previous sections. These differences occur because the sample has a finite number of seeds (30 or 100). It is assumed that seed selection is limited to the seeds in the sample without replacement; that is when a solution is not found a new seed will be selected from the seeds which remain and the seed(s) already used will not be put back into the seed pool. With this assumption, the event that a run successfully finds a solution in a given generation and the event that there are a given number of failed runs prior to a successful run are *not* independent.

For this section all the variables are defined as previously (section 7.3.4) with the exception that $G\mu$ and $T\mu$ represent the mean for the sample rather than for the population of all possible seeds. In addition, let p_f be the probability that exactly f runs fail to find a solution before a run successfully finds a solution. This is not

$$\frac{F}{n} \text{ as before but rather } \left(\frac{C_f^F}{C_f^n} \right) \left(\frac{S}{n-f} \right) \text{ where } \frac{C_f^F}{C_f^n} \text{ is the probability of } f \text{ failures}$$

and $\frac{S}{n-f}$ is the probability of a successful run after exactly f failed runs. As shown in section 7.3.5

$G\mu$ = average number of generations for a successful run
+ average number of generations for failed runs preceding the successful run.

Inserting the probability presented above yields

$$G\mu = G_{AV} + 1 + \sum_{f=0}^F \left(\frac{C_f^F}{C_f^n} \right) \left(\frac{S}{n-f} \right) [f(maxgen + 1)]$$

$$= G_{AV} + 1 + S(maxgen + 1) \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n}.$$

The index of summation has been changed from 0 to 1 because when $f=0$, the term is 0. (This change will also be made in the development of the formulas for the variance.)

$$\text{Similarly } T\mu = T_{AV:S} + T_{AV:F} S \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n}.$$

The second term of $T\mu$ should actually be calculated using the values that were averaged to get $T_{AV:F}$. Note that $T_{AV:F}$ is not a constant primarily due to system resources available at any given moment. Fortunately the variance of $T_{AV:F}$ is very small. (See figure 7.1 of section 7.1 for sample values.) Consequently it is not unreasonable to greatly simplify the calculations by treating $T_{AV:F}$ as a constant.

When a mean is the sum of the means of two random variables, as is the case for $G\mu$ and $T\mu$, $var(X\mu) = var(X_{AV:1}) + 2cov(X_{AV:1}, X_{AV:2}) + var(X_{AV:2})$ where $X\mu = X_{AV:1} + X_{AV:2}$. The covariance of two random variables, call them X and Y , is $\sum_i \sum_j x_i y_j p(x_i, y_j) - x_{AV} y_{AV}$ where $p(x_i, y_j)$ is the probability that X has value

x_i and Y has value y_j . When two events are independent the covariance is zero (Freund 1962). This is why the term with the covariance does not appear in the formula developed in section 7.3.6.

When substituting X_μ with G_μ , $X_{AV:1}$ is G_{AV+1} , and $X_{AV:2}$ is

$$S(maxgen+1) \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n}. \text{ In the covariance the } x_i \text{ are the values of } g_i,$$

$i \in \{1, 2, \dots, S\}$, and the y_j are $j(maxgen+1)$, $j \in \{0, 1, 2, \dots, F\}$. When choosing from a finite set without replacement, as was assumed in this case, the events are not independent. Thus, the covariance term for the calculation of the variance of G_μ must be retained.

Using the formula $\text{var}(X) = \left(\sum_i p_i x_i^2 \right) - x_\mu^2$, where $x_\mu = \sum_i p_i x_i$ and p_i is the probability associated with x_i (Freund 1962), the variance for the number of generations for failed runs

$$\begin{aligned} &= \sum_{f=0}^F \left(\frac{C_f^F}{C_f^n} \right) \left(\frac{S}{n-f} \right) [f(maxgen+1)]^2 - \left(S(maxgen+1) \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right)^2 \\ &= S(maxgen+1)^2 \left[\sum_{f=1}^F \frac{f^2 C_f^F}{(n-f) C_f^n} - S \left(\sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right)^2 \right]. \end{aligned}$$

Therefore

$$\begin{aligned} \text{var}(G_\mu) &= \text{var}(G_{AV+1}) + 2\text{cov}(G_{AV+1}, \text{number of generations for failed runs}) \\ &\quad + \text{var}(\text{number of generations for failed runs}) \end{aligned}$$

$$= \text{var}(G_{AV}+1) +$$

$$2 \left[\sum_{i=1}^S \sum_{f=0}^F g_i f (\text{maxgen}+1) \left(\frac{C_f^F}{C_f^n} \right) \left(\frac{1}{n-f} \right) - (G_{AV}+1) S (\text{maxgen}+1) \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right]$$

$$+ S(\text{maxgen}+1)^2 \left[\sum_{f=1}^F \frac{f^2 C_f^F}{(n-f) C_f^n} - S \left(\sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right)^2 \right].$$

$$= \text{var}(G_{AV}+1) +$$

$$2(\text{maxgen}+1) \left[\sum_{i=1}^S \sum_{f=1}^F \frac{g_i f C_f^F}{(n-f) C_f^n} - (G_{AV}+1) S \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right] +$$

$$S(\text{maxgen}+1)^2 \left[\sum_{f=1}^F \frac{f^2 C_f^F}{(n-f) C_f^n} - S \left(\sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right)^2 \right].$$

Replacing $(G_{AV}+1)$ with $T_{AV:S}$, $(\text{maxgen}+1)$ with $T_{AV:F}$, and g_i with t_i will yield the formula for $\text{var}(T\mu)$ which is

$$\text{var}(T_{AV:S}) + 2T_{AV:F} \left[\sum_{i=1}^S \sum_{f=1}^F \frac{t_i f C_f^F}{(n-f) C_f^n} - T_{AV:S} S \sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right]$$

$$+ T_{AV:F}^2 S \left[\sum_{f=1}^F \frac{f^2 C_f^F}{(n-f) C_f^n} - S \left(\sum_{f=1}^F \frac{f C_f^F}{(n-f) C_f^n} \right)^2 \right].$$

Unfortunately implementing these formulas for a sample also yields very high values for the variances. In addition, as pointed out earlier, the samples are not normally distributed. Consequently, it is also not practical to use these formulas to calculate values for standard statistical tests.

With these formulas also failing to provide a way to rank the methods for efficiency using standard statistical tests, the literature was again reviewed and other designs for ranking the methods were drawn up. This yielded the strategies which were ultimately implemented to determine the relative efficiency of the different methods. These strategies are described in the next section (7.3.10).

7.3.10 Ranking criteria

Because the high variances resulting from the incorporation of the failure rate into the calculations makes the use of standard statistical analysis impractical, a different basis for analysis of the data is needed. In the next subsection (7.3.10.1), the criteria, which are implemented in chapters 8 and 9, are presented. Program code for one of the criteria appears in subsection 7.3.10.2. Subsection 7.3.10.3 then summarizes the ranking criteria, and explains the use of the *U test* (Freund 1962) to obtain a degree of confidence for the results of the ranking procedures.

7.3.10.1 The ranking procedures implemented

With the attempts to use standard statistical analysis proving fruitless for the purposes of this study (sections 7.3.8 and 7.3.9), the literature was again reviewed to

search for different paradigm. When Angeline and Pollack (1993) compared the effectiveness of their compress and expand operators against a benchmark case, they executed five runs for each of the two methods. One run for each method failed to find a solution by generation *maxgen*. The four successful runs for each method were ordered based on the number of generations each took to find a solution. A pairwise comparison was made; that is the two best, the two second best, the two third best, and the two worst performers for each method were compared. Speed-up was defined as the number of generations for the benchmark divided by the number of generations for their new method. Their pairwise comparison yielded speed-ups between 1.2 and 3.0. They wrote that "each of the [paired] runs show speed-up in favor of compression" (p. 7). This comparison methodology forms the basis for part of the analysis presented in sections 8.2.2 and 8.2.3.

The pairwise comparison used by Angeline and Pollack (1993) is not directly applicable to our data because different failure rates appear across the methods within each experiment. However the concept of using speed-up or percent increase of one method over another (generally comparing each method to the benchmark) as a basis for evaluation is utilized. Hence it was decided to order or rank the methods (benchmark, MTF, SFS) based on different criteria (failure rate, G_{AV} , and G_{μ} for example) and to look for consistency across the rankings. In addition the values of the speed-ups or percent increases for the selected criteria are considered.

While the different formulas for G_{μ} and T_{μ} and their variances (sections 7.3.5 through 7.3.9) are not useful for standard statistical testing, they do point to the fact

that the failure rate is critical to these values and hence to the efficiency of each method. In addition, while Koza (1993) does not compare methods, he measures performance with $K = \frac{\log(1-z)}{\log\left(1-\frac{S}{n}\right)}$; with probability z , Koza expects to execute K independent runs to find a solution to a problem. This is based on an experiment which gets S successful runs out of n runs. K increases as $F=n-S$ increases. This is a further indication of the importance of the failure rate to the efficiency of a method. Since for a given experiment with a fixed *maxgen* K increases with F , the ordering or ranking of the methods based on either K or F will be the same. Since the focus is the ranking of the methods, not the actual values used for the ranking, ranking need only be done based on one of these two values, but K has an advantage over the failure rate because K can be used to determine speed-up as defined by Angeline and Pollack (1993) using the number of runs. Therefore K will be used as one of the criteria.

To evaluate average performance it was decided to rank the methods based on the increasing number of failures, increasing values of G_{AV} , and increasing values of $T_{AV:S}$ for each *maxgen* considered. While the two different $G_{\mu s}$ and $T_{\mu s}$ (for the population of seeds and for the sample) are not practical for standard statistical testing, they are calculated and also used to induce a ranking. When applicable the speed-up in terms of the benchmark is also calculated.

The main point of consideration for the evaluation purposes of this thesis is that the exact values of the selected parameters are not of critical importance; what is important is how the different methods are ranked based on the output of the

runs. Since there are numerous experiments, this thesis will focus on the individual samples and evaluate whether the ranking of the methods for the different experiments showed enough consistency.

Unlike the results of Angeline and Pollack (1993), the failure rate varied greatly across the methods within each experiment; therefore it was decided to do a groupwise comparison as suggested by Angeline and Pollack using only the best five runs for each method, best based on the number of generations needed to find a solution. Since different methods require different amounts of processor time to process a generation, the methods are similarly ranked based on a groupwise comparison of each of the best five runs using the processor times when the timing data is available. Due to the wide range of processors used for testing, consistent timing data is available for only one of the experiments.

As a separate criterion, machine size was also considered. Based on schema theory (sections 2.2.4 and 4.2) it is suspected that MTF will tend to find smaller FSMs. To test this hypothesis the top five runs based on machine size of successful FSMs are also compared groupwise to induce a ranking amongst all methods, but significant local differences do not manifest in the best 5 runs. Therefore additional runs from the list of each method are grouped. Consequently, for machine size, the i^{th} runs are grouped for ranking as long as all methods being compared have a successful i^{th} run. In addition the methods are ranked based on machine size averaged over the successful runs for a given method within an experiment.

While the rankings based on the two different $G_{\mu s}$ and $T_{\mu s}$ (sections 7.3.5,

7.3.7, and 7.3.9) will be examined, another ranking is utilized based on the number of generations (or amount of processor time) needed for a successful run following some unsuccessful runs. It was the desire to combine the data from the successful and unsuccessful runs which prompted the development of the $G\mu$ and $T\mu$ formulas in the first place.

To combine failure data, success data, and a pairwise comparison between runs within different methods as a basis for ranking, the following experiment design is assumed:

- 1) Randomly select one of the seeds from the complete set of seeds used for the experiment.
- 2) If the run fails to find a solution, select another seed from the set; use selection without replacement.
- 3) Repeat step 2 until a solution is found. This assumes that the set of seeds produced at least one successful run.
- 4) Repeat steps 1 through 3 for each method.

Assume that f of the n seeds are selected before a successful run finds a solution in generation g . The probability of this occurring is

$$\left(\frac{C_f^F}{C_f^n} \right) \left(\frac{\text{count}}{n-f} \right),$$

where count is the number of the S successful runs which found a solution in generation g . The methods are then ranked on the probability that a method will need fewer generations (or less processor time) to find a solution. The program

which ranks the methods for this appears in section 7.3.10.2.

7.3.10.2 Code for probability of getting a solution first

The last of the criteria used to induce a ranking can best be explained by examining the program which produces the ranking. This program appears in figure 7.2.

This program is to be used for an experiment in which a GA is executed for different methods. Each method is executed for n distinct runs. S of these n runs locate a solution in generations $g_1, g_2, \dots, g_S$ with corresponding processor times $t_1, t_2, \dots, t_S$. F of the runs fail to find a solution before generation *maxgen*.

```

line#  /* rank by % of runs which get results sooner */
1:      #include <stdio.h>
2:      #include <math.h>

3:      #define num_of_methods 14
4:      #define run_name "test"

        /* float x[2550][num_of_methods], templ; */
5:      int x[2550][num_of_methods], templ;

6:      typedef char method_name[4];

7:      method_name method[14] = { "M2  ", "M1  ",
8:          "M2-2", "M1-2", "M2-3", "M1-3", "M2-4", "M1-4", "M2-5",
9:          "M1-5", "S2  ", "S1  ", "B2  ", "B1  "};

10:     int num_of_rows[num_of_methods];
11:     double px[2550][num_of_methods];
12:     float pct_inc[num_of_methods];
13:     FILE *fp_out;

```

Figure 7.2: Ranking Based on f Failures Followed by a Successful Run. Continued on next page.

```

                                /*****/

14: void xchange_cols(int j)

15: { int i, maxi;
16:   double temp2;
17:   char temp3;

18:   if (num_of_rows[j-1] >= num_of_rows[j])
19:     maxi = num_of_rows[j-1];
20:   else maxi = num_of_rows[j];

21:   for (i=0; i<maxi; ++i) {
22:     temp1 = x[i][j]; x[i][j] = x[i][j-1]; x[i][j-1] = temp1;
23:     temp2 = px[i][j]; px[i][j] = px[i][j-1]; px[i][j-1] = temp2;
24:   }
25:   for (i=0; i<4; ++i) {
26:     temp3 = method[j][i];
27:     method[j][i] = method[j-1][i];
28:     method[j-1][i] = temp3;
29:   }

30:   temp2 = pct_inc[j];
31:   pct_inc[j] = pct_inc[j-1];
32:   pct_inc[j-1] = temp2;
33:   i=num_of_rows[j];
34:   num_of_rows[j]=num_of_rows[j-1];
35:   num_of_rows[j-1]=i;
36: } /* xchange_cols */

                                /*****/

37: void main(void)

38: { int i, j, k, left, right;
39:   double sum_left, sum_right;

/* put values into x[][], px[][] & num_of_rows[] */

/* actual main starts here */

40:   fp_out = fopen(run_name ".rnk", "w");
41:   fprintf(fp_out, "%s\n", run_name ".rnk");

```

Figure 7.2 continued: Ranking Based on f Failures Followed by a Successful Run. Continued on next page.

```

42:     for (i=1; i<num_of_methods; ++i) {
43:         sum_left = 0; sum_right = 1;
44:         for (j=i; j>0 && sum_left<sum_right; --j) {
45:             sum_left=0; sum_right=0;
46:             for (left=0; left<num_of_rows[j-1]; ++left)
47:                 for (right=0; right<num_of_rows[j]; ++right)
48:                     if (x[left][j-1] < x[right][j])
49:                         sum_left += px[left][j-1] * px[right][j];
50:                     else if (x[left][j-1] > x[right][j])
51:                         sum_right += px[left][j-1] * px[right][j];
52:                     if (sum_left >= sum_right)
53:                         pct_inc[j-1] = 100 * (sum_left - sum_right) / sum_right;
54:                     else {
55:                         pct_inc[j] = 100 * (sum_right - sum_left) / sum_left;
56:                         xchange_cols(j);
57:                     }
58:             } /* for j */
59:         } /* for i */
60:         fprintf(fp_out, "      %% increase\n");
61:         for (k=0; k<num_of_methods-1; ++k)
62:             fprintf(fp_out, "%c%c%c%c %13.8e%%\n",
63:                 method[k][0], method[k][1], method[k][2], method[k][3],
64:                 pct_inc[k]);
65:         fprintf(fp_out, "%c%c%c%c\n",
66:             method[k][0], method[k][1], method[k][2], method[k][3]);
67:         fclose(fp_out);
68:     } /* main */

```

Figure 7.2 continued: Ranking Based on f Failures Followed by a Successful Run

When the ranking is based on the number of generations, the values being compared (that is the number of generations evaluated to get a solution) is an integer. When the values being compared represent the amount of processor time needed to locate a solution the values are real/floating point. Look at line 5 of figure 7.2 and the line immediately before it. One of these two lines must be commented out depending on whether the values being compared are the number of generations or the amount of processor time. The `px` array on line 11 is the probability of the value stored in the corresponding location of the `x` array of line 5. For any given

successful run, the **x** array will contain the values $g_i+1, g_i+1 + \text{maxgen}+1, g_i+1 + 2(\text{maxgen}+1), \dots, g_i+1 + f(\text{maxgen}+1), \dots, g_i+1 + F(\text{maxgen}+1)$, $i \in \{1,2,\dots,S\}$ where g_i is the generation number in which a solution was found, or $t_i, t_i + \text{maxgen}+1, t_i + 2(\text{maxgen}+1), \dots, t_i + f(\text{maxgen}+1), \dots, t_i + F(\text{maxgen}+1)$, $i \in \{1,2,\dots,S\}$ where t_i is the processor time corresponding to g_i .

When there are 100 seeds, the maximum number of values which will have to be stored will occur when half of the 100 seeds result in failed runs. Therefore the maximum number of values which will have to be stored will be 50 values for successes $\times$ 51 values for failures (includes $f=0$) = 2550 values. This is reflected in the number of rows reserved in the **x** (line 5) and **px** (line 11) arrays and is based on the fact that a maximum of 100 seeds was used for the experiments for this thesis. The second index of the **x** and **px** arrays indicates the method. The value of the second index corresponds to the value of the index in the **method** (lines 7-9), **num_of_rows** (line 10), and **pct_inc** (line 12) arrays. **num_of_rows** is to contain the number of rows of data actually stored in the column of the **x** and **px** arrays for the corresponding method. The **pct_inc** array will be discussed later.

The **xchange_cols** function (lines 14-36) exchanges the values in two adjacent columns of the **x** and **px** arrays and all values associated with those columns. It is invoked (line 56) after the calculations are completed, and when as a result of the calculations it is determined that the methods in the two adjacent columns are out of order.

The data, which are to be placed in the **x** and **px** arrays of the **main**

procedure (lines 37-68), is stored on different files. A filter must take the relevant values from the proper files and fill the **x** array with either the $g_i + 1 + f(maxgen + 1)$ or $t_i + f(maxgen + 1)$ values, $i \in \{1, 2, \dots, S\}$, $f \in \{0, 1, 2, \dots, F\}$. (See the comment line after line 39.) The filter has to collect the data from the files created by the execution of the GA for the different methods, fill the **x** array with the appropriate values from the files for each method in the experiment, and reflect the amount of data in each row of the **x** array in the corresponding location of the **num_of_rows** array. The filter also has to calculate the values for the **px** array. This probability is

$$\frac{C_f^F}{C_f^n} \left(\frac{1}{n - f} \right),$$

where f is the number of failed runs associated with the corresponding value in the **x** array. After the filter places the collected and computed values into the program (See comment between lines 39 and 40 of figure 7.2.), the filter must compile and execute the program.

Once the data is available, **main** opens the file which will receive the output (lines 40-41). Then the methods in adjacent columns are systematically compared. Each method is eventually moved into its proper rank position. The **i** loop (lines 42-59) indicates the method which is being moved into its proper rank position in the list of methods before it. The instructions in the **j** loop (lines 44-58) move this method to its proper position.

The i^{th} method as indicated in the **method** array is always compared to the method in the column to its immediate left. **sum_left** accumulates the probabilities

when the value for the method in the $j-1^{\text{st}}$ column of the **x** array is smaller than the value in the j^{th} column of the **x** array (lines 48-49). **sum_right** similarly accumulates the probability for the right or j^{th} column (lines 50-51). These accumulators are initialized on line 43 so as to permit the **j** loop to start. When **sum_left** is less than **sum_right** (line 44) it means that as a result of the most recent column comparison the two columns which were compared were out of order. This indicates that the i^{th} method has to be moved one column to the left and still has to be tested to see if is in its proper ranked position. When **sum_right** < **sum_left**, the i^{th} method is more likely to require more generations (or processor time) to find a solution and it is therefore in its proper ranked position. Once the i^{th} method is in its proper position, it is no longer necessary to continue comparing this method to others. This is why **sum_left** < **sum_right** (line 44) is one of the criteria for continuing the **j** loop; but, this necessitates initialization of **sum_left** and **sum_right** (line 43) prior to entering the loop to guarantee that the loop will start execution.

The **j** loop moves the i^{th} method along by comparing it to the method in the column to its immediate left. Actually the **i** and **j** loops control a sort, but the values needed for the comparison of the sort must be recalculated for each comparison. The values determining the sorting process are not fixed for each method but determined by a competition between the methods. **sum_left** and **sum_right** are the values which are compared for the sort. **sum_left** and **sum_right** measure the relative performance of two methods. Each measures the

probability that the method of the corresponding column will find a solution before the method of the other column.

The accumulators, **sum_left** and **sum_right**, are cleared on line 45. Then each value (number of generations or amount of processor time) in the **left** column is compared with each value in the **right** column. This is accomplished with the **left** and **right** loops (lines 46-51). The column with the lower value in the **x** array (fewer generations or less processor time to a solution) is desired; therefore it is credited with the probability of the intersection of the two events (lines 48-51), that is with the product of the probabilities of the two events. It is felt that the relative performance of two adjacent methods should be defined by how much better the more efficient of the two methods is. Therefore the percent increase of the better method is calculated as $100\left(\frac{\text{larger sum} - \text{smaller sum}}{\text{smaller sum}}\right)$, and then stored in **pct_inc** (lines 52-57). The two methods being compared are in their proper positions with respect to each other when the one to the left has a higher probability of getting results sooner (**sum_left** > **sum_right**), or when their performance is the same (**sum_left** = **sum_right**). When this occurs it is only necessary to store the **pct_inc** for the method in column **j-1** (lines 52-53). When the **ith** method (which at this point is stored in the **jth** column) is the more efficient of the two, it is necessary to store the **pct_inc** of the better method and move this method into its proper position (lines 54-57).

Once the sort is completed, the ranking is output (lines 60-66) and the output file is closed (line 67). When applicable, the percent increase over the benchmark

is also calculated. This does not necessarily correlate exactly to the ranking previously described. There are instances when one method is ranked better than another based on the program in figure 7.2, but the ranking when compared to the benchmark reverses that order. These instances are few.

N = 5 runs/method
maxgen = 50

Method A
 F = 2 failures

f	0	0	0	1	1	1	2	2	2
#gen	15	25	45	66	76	96	117	127	147
prob	.2	.2	.2	.1	.1	.1	.33333	.33333	.33333
prob better	.2	.08	.04	.02	.005	0	0	0	0

sum_left = .345

Method B
 F = 1 failure

f	0	0	0	0	1	1	1	1
#gen	20	23	25	35	71	74	76	86
prob	.2	.2	.2	.2	.05	.05	.05	.05
prob better	.16	.16	.12	.12	.015	.015	.01	.01

sum_right = .61
%increase = 76.8

Figure 7.3: Calculation of Percent Increase

A simple example will clarify the calculation of the percent increase of one method over another. Figure 7.3 contains data for two methods. As indicated in the figure, there were five runs for each of the two methods. Each run which failed to find a solution processed $\text{maxgen} + 1 = 51$ generations (the initial generation plus the 50 generations bred). For method A, two of the five runs failed to find a solution

by generation 50, and one of the five for method B did not locate a solution after processing 51 generations. $\mathbf{f}$ is the number of failed runs preceding a successful run. Therefore when $\mathbf{f}=0$, the value in the **#gen** row represents the number of generations the GA processed for a single successful run. Hence, the three successful runs for method A found solutions after processing 15, 25, and 45 generations. Notice that adding 51 ($\mathbf{maxgen}+1$) to each of these numbers yields 66, 76, and 96, respectively—the numbers listed for a single failed run ($\mathbf{f}=1$) followed by a successful run. The **prob** is the probability that the GA will have to process **#gen** generations to get a solution assuming the seed pool consists of the five seeds used for the given method and seed selection without replacement. The **#gen** in each column for method A is compared to the **#gen** in each column for method B. Since method A has nine columns and method B has eight columns, there will be $9 \times 8 = 72$ comparisons. For example, one of these comparisons involves the 76 generations for method A which is compared to the 23 from method B. There is a .1 probability that method A will find a solution after processing 76 generations, one failed ($\mathbf{f}=1$) run and a single successful run, and a .2 probability that method B will find a solution after processing 23 generations for a single successful run ($\mathbf{f}=0$). Therefore the probability that method A will find a solution in 76 generations and method B in 23 generations is $.1 \times .2 = .02$; that is this situation is expected to occur 2% of the time. In this case, method B is more successful. The number in the **prob better** row is the summation of the probabilities when the given method and **#gen** prevail. A tie is credited to neither method. **sum_left** and **sum_right**, as they are denoted

in the program (figure 7.2), are the sums of the values in the **prob better** column for the given method. Since **sum_left** for method A is less than **sum_right** for method B, method B is more likely to get a solution before method A, based on the sets of seeds used for the experiment and seed selection without replacement. The percent increase of method B over method A is $100 \times \frac{.61 - .345}{.345} \approx 76.8\%$.

This section described the most complex of the criteria for ranking. In the next section (7.3.10.3) the ranking criteria are summarized.

7.3.10.3 Summary of ranking criteria

This section summarizes the criteria used to evaluate the data (chapter 8). The evaluation is based on rankings induced by different criteria. Rankings are induced by G_{AV} (average number of generations for a successful run to find a solution), $T_{AV:S}$ (processor time corresponding to G_{AV}), F (failure rate), and the two $G_{\mu s}$ and $T_{\mu s}$ (defined in sections 7.3.5, 7.3.7, and 7.3.9). In addition, the runs for each method are sorted based on the g_i and t_i which are used to calculate G_{AV} and $T_{AV:S}$ respectively. The i^{th} runs, $i \in \{1, 2, 3, 4, 5\}$, on the lists of the different methods are the basis for rankings. For these rankings the speed-up with respect to the benchmark is calculated. A ranking is also induced based on the probability that one method will get a solution before another (section 7.3.10.2). For this ranking the percent increase is calculated. While not a consideration with respect to efficiency per se, the size of the successful machines located by the GA are also ranked based on grouped runs and average machine size.

It is expected that the values used to rank the methods for a given criterion will vary greatly from experiment to experiment. As mentioned earlier, the concern is not the actual values collected and calculated, but rather the order that these values impose on the different strategies and whether there is some consistency in these rankings. The rankings will be compared, with the ranking based on machine size treated as a separate issue. The most desirable outcome will occur if one method consistently or almost consistently appears to be more efficient than the other methods. Lacking that, the next best result will occur if the two new operators (MTF, SFS), or one of the two new operators, consistently or almost consistently performs better than the benchmark.

The values for G_{AV} , $T_{AV:S}$, $G_{\mu s}$ and $T_{\mu s}$ for the sample, the probability that one method will get a solution before another, and average machine size are calculated using a finite number of points. Because a finite number of points is used for these calculations, a *U test*, also known as the Mann-Whitney test or the Wilcoxon test (Freund 1962), is applied after the best performing method(s) is/are selected. The *U test* is used to obtain a degree of confidence for the difference in performance between the selected method(s) and the benchmark to determine if this difference is *significant*, or merely due to chance. For the *U test*, two samples are treated as a single sample. Rank numbers are assigned to the combined list of measured values of a random variable. A *z* score is then computed using the rank numbers as opposed to the actual values of the random variable. To use the established table for the area under a standardized normal distribution (that is a normal distribution

with a mean of 0 and a standard deviation of 1) to determine a degree of confidence that the difference between the two samples is not due to chance, that is the difference is *significant*, each of the two samples must contain at least nine points. In a few of our experiments there were not enough data points (due to a high failure rate) to employ the *U test*.

Whether the rankings will be consistent across the different experiments or across the different bases for comparison (number of generations for a successful run, failure rate, processor time, or some combination of these) and whether the differences in performance are significant will not be known until the data is evaluated.

In the next chapter the data obtained during testing is combined with the evaluation strategies described in this chapter to evaluate the relative performance of the different methods.

>>>●<<<

Chapter 8

Evaluation of the algorithms

In this chapter the experiments conducted for this thesis are described, and the resulting data are analyzed using the criteria set forth in chapter 7. Two different types of data were collected: per generation data (section 8.1) and per run data (section 8.2). In section 8.1.1 the experiments were designed to produce the per generation data to track the evolutionary process. These data were then graphed to evaluate the evolutionary process. The graphs along with the corresponding evaluation of the data appear in section 8.1.2. In section 8.2.1, the experiments which produced the per run data are described. These data are used to evaluate convergence/efficiency, and analyzed in sections 8.2.2 and 8.2.3. Section 8.3 contains the conclusions obtained from analyzing both sets of data.

8.1 Evaluation of the evolutionary process

Before looking at the data used to evaluate the evolutionary process, the experiments and data collection are described in the next subsection (8.1.1). Then in the following subsection (8.1.2) the data are evaluated.

8.1.1 The experimentation

The experiments for this thesis were carried out with different population sizes. One set of experiments followed the example of Jefferson et al. (1992) and used a population size of 64K FSMs. For these experiments the last generation created (call this *maxgen*) is 70. A similar set of experiments were conducted with a population size of 16K FSMs and a *maxgen* of 125.

The purpose of this first set of experiments is to permit tracking of the evolutionary process. Consequently, *maxgen* generations were bred for each run, even when a perfect scoring FSM appeared before that number of generations were completed. Fourteen methods were executed for each population size. The methods are labelled as follows:

- 1) The first letter(s) is(are) B (benchmark), S or SFS (SFS algorithm added to benchmark), or M or MTF (MTF algorithm added to benchmark). Every generation of a given run had the same method applied throughout the entire run.
- 2) The letter code is followed by a 1 (no competition) or 2 (competition included). Competition was also applied to every generation of a given run.
- 3) If a q(arter) follows the 1 or 2, the population contains 16K FSMs. When the q is omitted the population size is 64K (full).

During the testing/debugging of the programs it became obvious that the effects of reorganization carried over to some extent from one generation to the next. In addition, initial testing indicated that MTF outperformed SFS. These two conditions prompted the consideration of a variant application of MTF. For these runs the MTF algorithm was applied to the initial parent pool, and then to every i^{th} generation after that. These runs are labelled M1- i and M2- i , $i \in \{2,3,4,5\}$.

Using this described nomenclature, the $2 \times 14 = 28$ sets of runs executed for the first part of the study are B, S, and M with/without competition, and with MTF applied in rotating fashion for full (64K FSMs, left side) and quarter (16K FSMs, right side) population sizes:

B1, B2, S1, S2, M1, M2,		B1q, B2q, S1q, S2q, M1q, M2q,
M1-2, M1-3, M1-4, M1-5,	and	M1q-2, M1q-3, M1q-4, M1q-5,
M2-2, M2-3, M2-4, M2-5,		M2q-2, M2q-3, M2q-4, M2q-5.

30 runs were executed for each method or variant. Each set of 30 runs used the same set of 30 seeds for the random number generator; this set of seeds appears in section 4.3. Run number i across the methods used the same seed, and hence started with the identical initial population for a given population size. For this set of experiments the data printed out were for the purpose of providing information on the evolutionary process. Statistical data were recorded for every generation as indicated in figure 4.25 (section 4.3), and figures 5.8 and 5.9 (section 5.3). The results of these runs are discussed in the next section.

8.1.2 Data analysis

The effect of the different methods on the evolutionary process is evaluated by observing a series of graphs and evaluating how the GA progresses from generation 0 to generation *maxgen*, the loop limit. One set of graphs contains individual runs. Another set contains results averaged over the 30 runs executed for each method. Based on these graphs MTF performs better than the benchmark and SFS.

8.1.2.1 B1, MTF1, and SFS1 compared

In the first evaluation, B1 (benchmark), MTF1, and SFS1 are compared. The "1" in the name of each method indicates that competition was not used. For each experiment (sections 8.1.2.1.1 and 8.1.2.1.2) the reader is first referred to the graphs in figures C.1a (population size 64K) and C.2a (population size 16K) in appendix C. On these graphs, runs are compared based on the same initial population: that is each graph contains runs which start with the same individuals in generation 0. It is shown that for both the 64K and 16K populations the MTFs perform the best.

8.1.2.1.1 64K population

In figure C.1a the population consists of 64K FSMs. In 12 of the 30 runs (run numbers 2, 4, 12, 15, 16, 18, 19, 22, 24, 25, 26, and 27) MTF1 is clearly the most effective since the line representing MTF1 tends to rise above the other two methods. While MTF1 seemed to perform better than the other 2 methods on run 17, it did not find a perfect scoring ant. In addition in runs 7, 9, 14, 21, and 24 MTF1 is the first to locate a solution. In run 11 MTF1 and B1 appear to have converged prematurely. In runs 10, 13, and 23 SFS1 is the first to find a solution. In addition SFS1 performs best in runs 5, 11, 20, and 29. The benchmark (B1) is the first to find a solution in runs 1, 6, 8, and 28, but on the whole does not show a tendency to rise above the other two methods.

Figure C.1b presents the same 30 runs compared by evolutionary progress. (See section 7.1 for the description of how the runs were selected for each graph.)

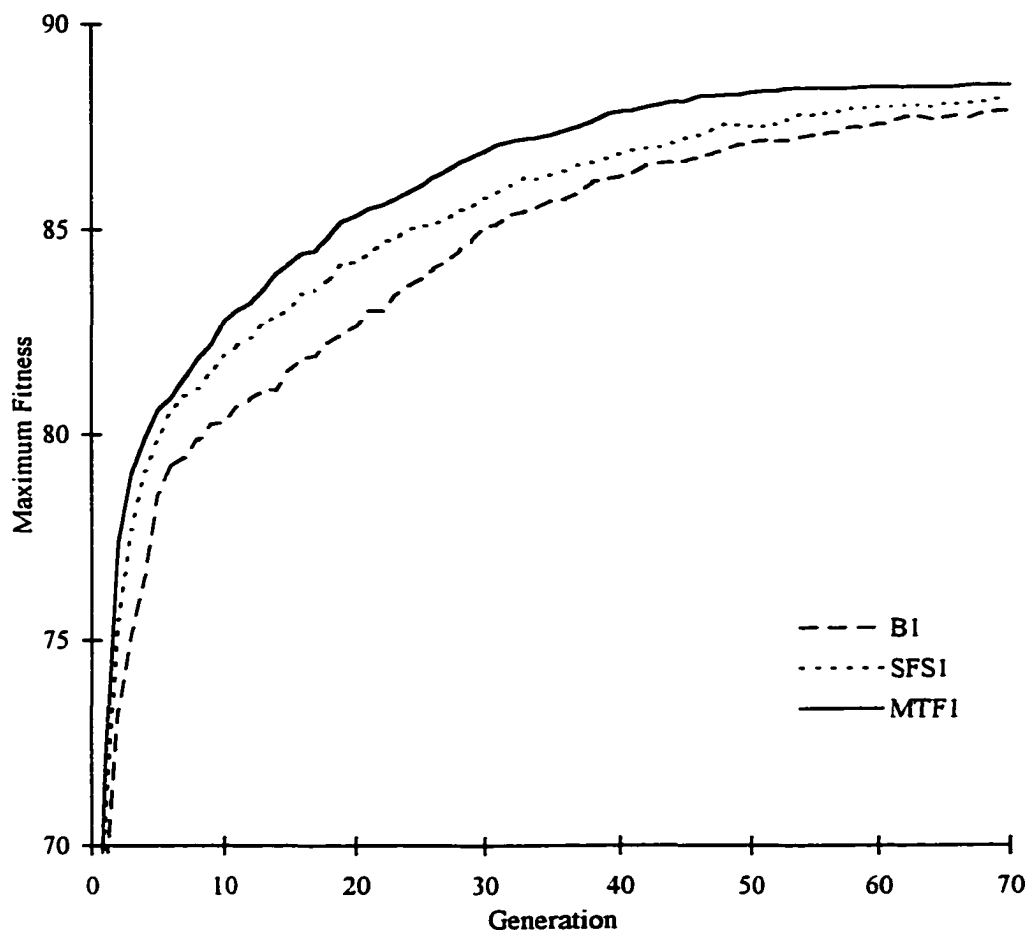


Figure 8.1: Comparison of B1, SFS1, and MTF1. Averages over 30 runs of maximum fitness in each generation. Population size 64K.

On 13 of these graphs (graphs 5, 6, 7, 8, 9, 10, 18, 19, 22, 23, 24, 27, and 29) MTF1 is above the other methods in a significant part of each graph., and on graphs 1, 2, 3, 4, 11, 12, 13, 15, 16, 17, 25, 26, and 30 it is the first to find the highest fitness located. MTF1 did not find a perfect scoring ant on the graphs with italicized numbers. In figure C.1c the same data are repeated, but here the comparison is based on the location of the optimal individual. MTF1 is the first to locate the highest fitness found by the runs appearing on each graph except on graph 28.

MTF1 does not find a perfect scoring FSM on graphs 23-30.

The maximum fitnesses found in each generation were then averaged over the 30 runs for each method. These averages appear in figure 8.1. On the average, MTF1 clearly outperforms the other two methods.

8.1.2.1.2 16K population

Figures C.2(a-c) also compare runs of B1, MTF1, and SFS1, but these runs utilize a population of 16K FSMs. Figure C.2a compares runs which have the same initial population. In runs 1, 2, 4, 5, 6, 7, 9, 10, 12, 13, 18, 21, 24, 25, 26, and 29, MTF1 is clearly superior, while in runs 11, 14, 16, 22, 23, and 30 it is the first to locate the highest fitness which appears on each graph. An optimal FSM is not found in the italicized runs. In some runs one method appears to be the best at the start, but its performance is then surpassed by another method. For example in runs 8 and 27, MTF1 starts off ahead but in the end B1 surpasses it. Also in run 20, MTF1 also jumps ahead, but it quickly converges prematurely and is then surpassed by both SFS1 and B1, with SFS1 doing slightly better than B1. SFS1 looks best only in run 19 and is first to find a solution in run 20. The benchmark is the first to find the highest fitness which appears on each graph in runs 3, 8, 15, 17, 27, and 28.

Figure C.2b compares the same runs based on evolutionary progress. On 21 of the 30 graphs (graphs 2-5, 7, 9, 15-18, and 20-30), MTF1 stays above the other two methods during a significant part of the process while on the remaining graphs it is the first to find the highest fitness which appears on each graph. In figure C.2c

comparisons are based on the location of the optimal individual. On all of these graphs MTF1 is the first to find the highest fitness which appears on each graph. Figure 8.2 contains the average performance for each of the three methods with a population size of 16K. Averages are taken over the 30 runs for each method. Again here MTF1 performs better than the other two methods.

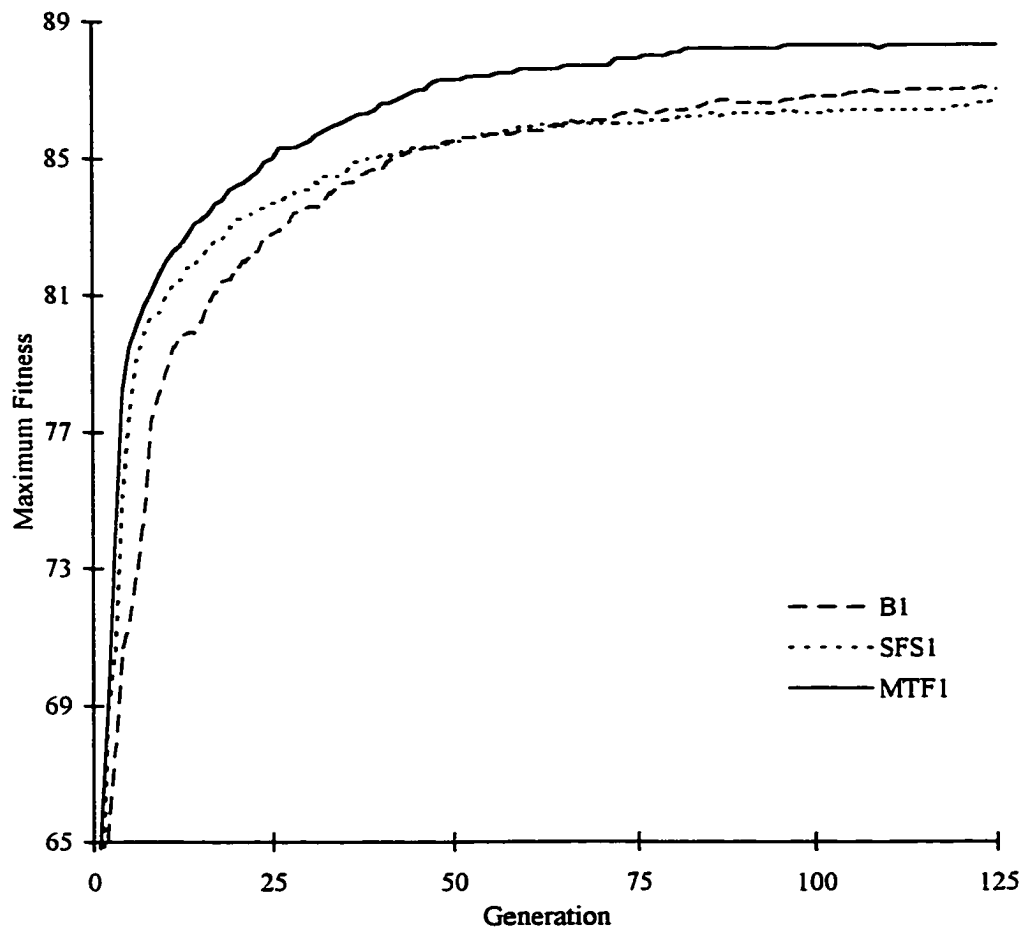


Figure 8.2: Comparison of B1, SFS1, and MTF1. Averages over 30 runs of maximum fitness in each generation. Population size 16K.

8.1.2.2 B and SFS compared with/out competition

Figures C.(3-4)(a-c) compare B1, B2, SFS1, and SFS2. Recall that the "1" following the method name means that competition was not employed, while a "2" signifies that competition was implemented to augment the fitness function for the selection process. Based on average performance, the SFSs start off stronger than the benchmarks in the early generations, but they do not maintain this advantage as the runs progress towards a solution.

Figures C.3(a-c) represent runs with a population size of 64K FSMs, while figures C.4(a-c) are for runs with a population of 16K individuals. Figures C.(3-4)a compare runs with the same initial population. Figures C.(3-4)b base the comparison on evolutionary progress, and figures C.(3-4)c use the location of the optimal individual as the basis for comparison. All of these graphs appear in appendix C and are summarized on the graphs in figures 8.3 and 8.4. Figure 8.i contains averages over the 30 runs for the graphs in figures C.i(a-c), $i \in \{3,4,5,6\}$. The remainder of the data analysis is based primarily on these graphs presenting the averages. The graphs in appendix C are referenced when needed to enhance the explanation of the average performance.

In figure 8.3 (population size 64K) SFS1 (no competition) prevails only slightly for a significant portion of the runs, but at the end it is exceeded by B2 (implements competition). The goal of these methods is convergence to an individual with the optimal fitness in the fewest generations. For more insight as to why B2 prevails at the end, the reader is now referred to the graphs in figure C.3c on which the basis

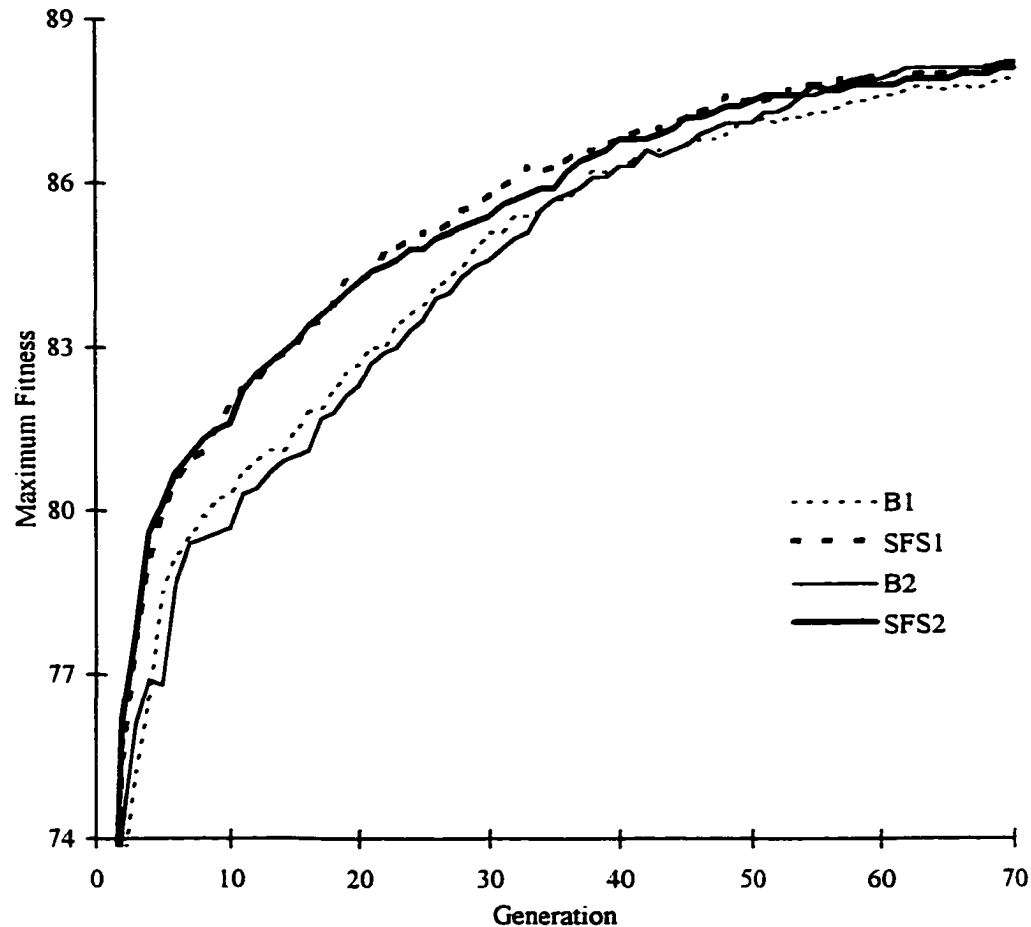


Figure 8.3: Comparison of B1, B2, SFS1, and SFS2. Averages over 30 runs of maximum fitness in each generation. Population size 64K.

for comparison is location of an optimal individual. SFS1 locates an FSM with the optimal fitness on graphs 1-18, SFS2 on graph 1-16, B1 on graphs 1-14, and B2 on graphs 1-20. On all but three graphs (graphs 27, 28, and 30) B2 matches or exceeds the highest fitness found by the four runs appearing on each graph, but it is generally not the first to locate that highest fitness. With B2 almost consistently matching or exceeding the highest fitness appearing on each graph, B2 on the average does better than the other three methods for a short time towards the end of the runs (figure

8.3), but for most of the time SFS1 prevails or matches the other methods.

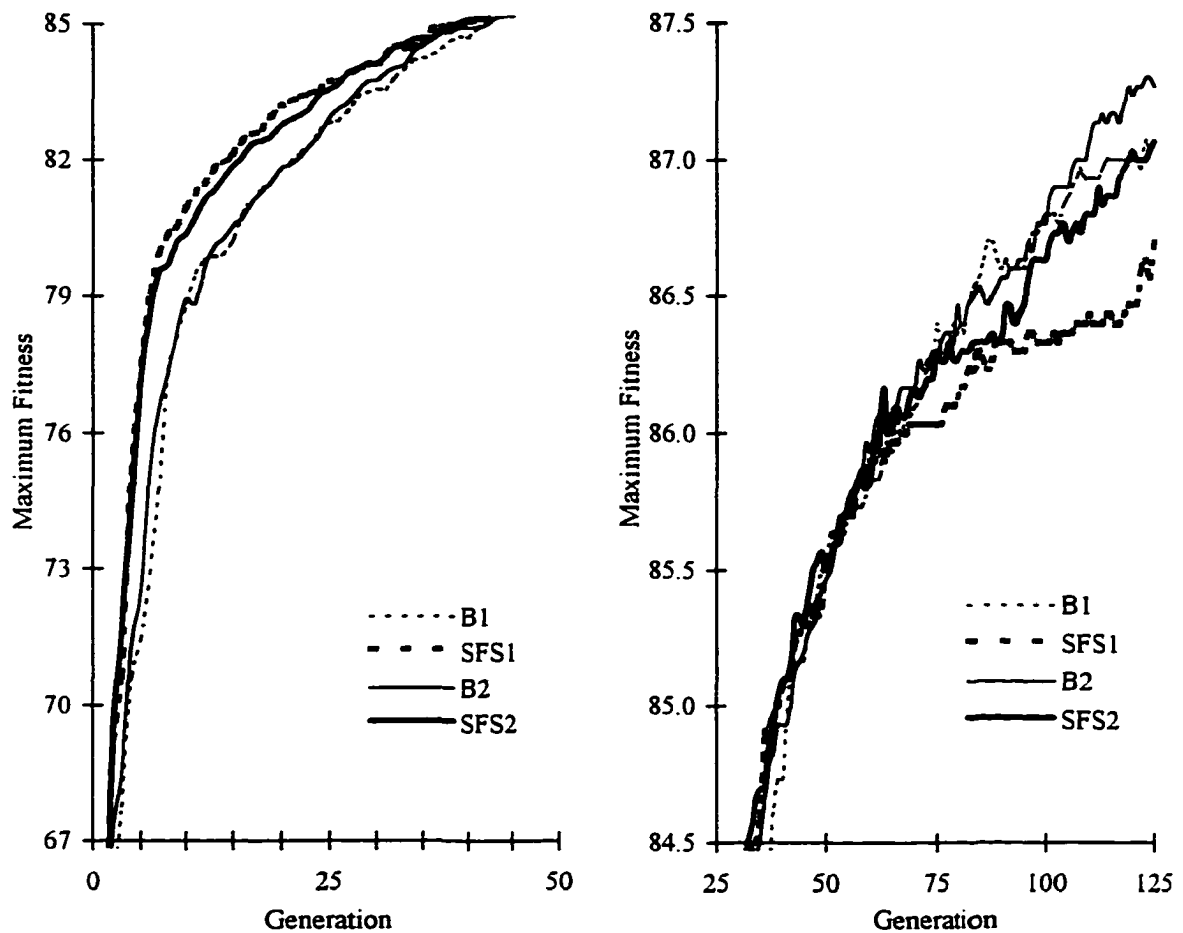


Figure 8.4: Comparison of B1, B2, SFS1, and SFS2. Averages over 30 runs of maximum fitness in each generation. Population size 16K.

In figure 8.4 (population size 16K) SFS1 (no competition) initially jumps ahead of the other three methods with SFS2 coming in a close second. But about halfway through SFS1 falls behind, as does SFS2 a bit later. In the end B2 prevails. A pattern similar to the performance of B2 in the 64K case exists. In figure C.4c of appendix C, B2 locates an FSM with optimal fitness in more runs than the other

three methods, and on all but two of the graphs (22 and 23) it matches or exceeds the highest fitness appearing on each graph. On graphs 3-8, 11-16 18, 20, 21, 25, 26, and 28, (18 of the 30 graphs), it is the first to locate the highest fitness appearing on each graph. The SFS algorithm does not seem to have any clear advantage over the benchmark.

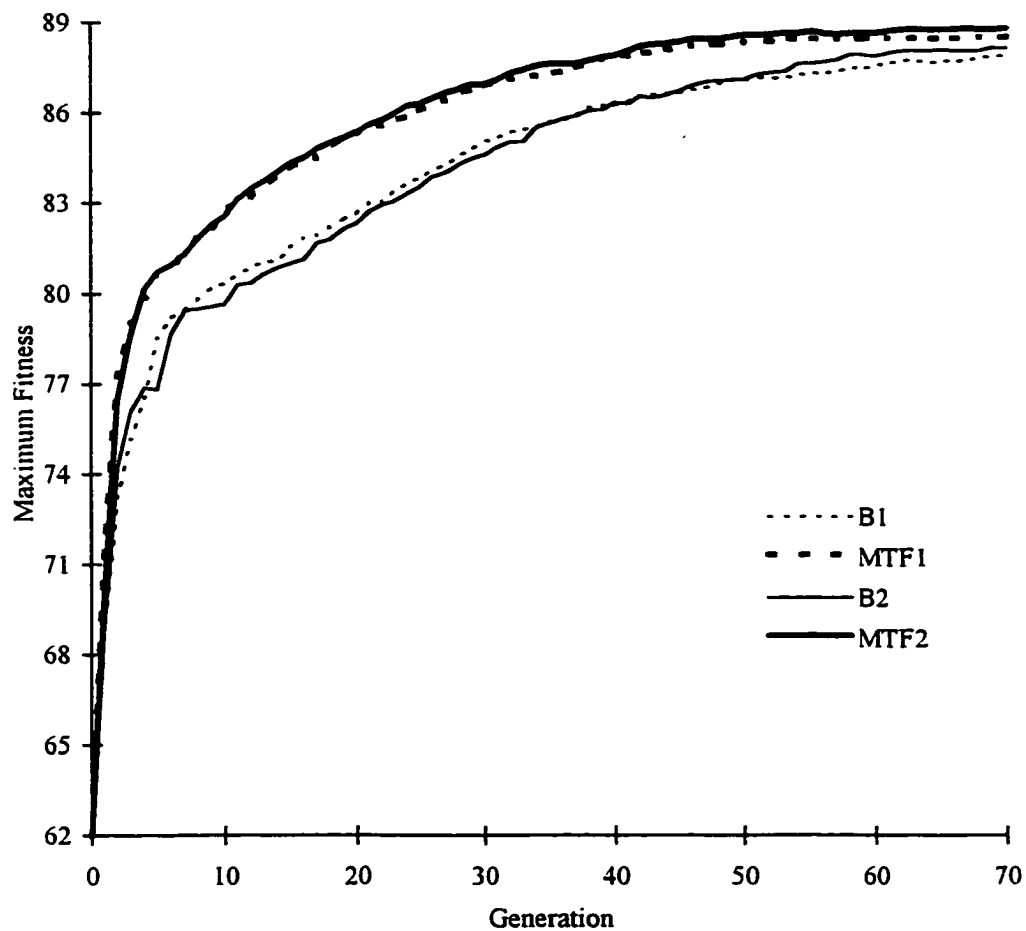


Figure 8.5: Comparison of B1, B2, MTF1, and MTF2. Averages over 30 runs of maximum fitness in each generation. Population size 64K.

8.1.2.3 B and MTF compared with/out competition

The next set of graphs, C.(5-6)(a-c) in appendix C, compare B1, B2, MTF1, and MTF2. Overall, the MTFs are superior to the benchmarks with MTF2 having a very slight advantage over MTF1.

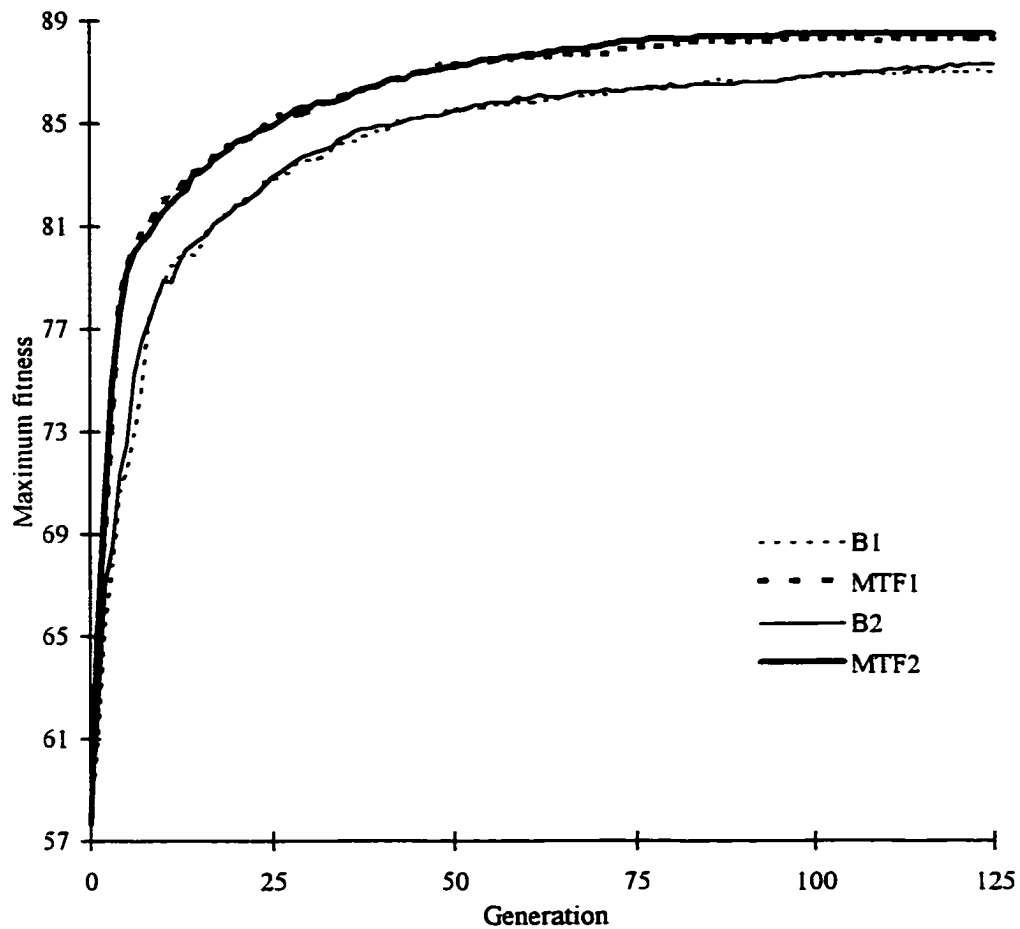


Figure 8.6: Comparison of B1, B2, MTF1, and MTF2. Averages over 30 runs of maximum fitness in each generation. Population size 16K.

C.5(a-c) contains the runs which have a population of 64K individuals, while for C.6(a-c) the runs utilize a population size of 16K. As for the comparisons in

C.(1-4)(a-c), C.(5-6)a compares runs with identical initial populations, C.(5-6)b employs the evolutionary process as the basis for comparison, and C.(5-6)c compares runs based on the location of a solution. The average performance over the 30 runs appears in figures 8.5 (64K population size) and 8.6 (16K population size). Both of these graphs indicate the superiority of MTF over the benchmark, with MTF2 (with competition) having a slight edge over MTF1 (no competition).

Based on the graphs in figures 8.1 through 8.6 and C.(1-6)(a-c), MTF seems to be superior to both SFS and the benchmark. Based on these experiments MTF appears to move the GA ahead faster with respect to convergence. Based on figures 8.5 and 8.6, adding competition to MTF gives the process a slight boost.

8.1.2.4 Periodic reorganization

When the programs which generated the data of the previous subsections were being tested, the reorganization of the genomes seemed to carry over to some degree from generation to generation: that is, the offspring of parents which were reorganized by SFS or MTF frequently do not need a great deal of reorganization by the same operator. This raises the question as to whether it is necessary to reorganize the parent pool every generation in order to gain the benefits of reorganization.

Since MTF appears to be superior to SFS, runs to determine whether there is an advantage to reorganizing the genomes in the parent pool every generation were executed only with the MTF algorithm. MTF, both with (MTF2) and without

(MTF1) competition, was applied to the parent pool every generation (MTF1 and MTF2), every other generation (MTF1-2 and MTF2-2), every third generation (MTF1-3 and MTF2-3), every fourth generation (MTF1-4 and MTF2-4), and every fifth generation (MTF1-5 and MTF2-5). When competition was employed, it was applied in every generation.

The graphs in figures 8.7 through 8.10 compare the MTF runs, with MTF applied every i^{th} generation, $i \in \{1, 2, 3, 4, 5\}$. The runs were executed for the two different population sizes both with and without competition. The graphs contain the averages of the maximum fitness found in each generation. These averages are taken over the 30 runs executed for each combination of population size/generation gap for application. On three of these four graphs there is a clear but slight advantage to applying MTF every generation. The same data are presented in figures 8.11 and 8.12, but on these graphs MTF1 and MTF2 are compared. As was noted earlier, MTF2 has a slight edge over MTF1 when MTF is applied every generation. On six of the ten graphs appearing in these two figures, MTF1 seems to have a slight edge during the central part of each set of runs, but as the generation approaches the last generation bred neither with nor without competition seems to have a clear advantage.

Based on all the graphs discussed in this section, SFS does not have any consistent advantage over the benchmark. MTF performs better than the benchmarks and SFSs, with MTF2 having a very slight advantage over MTF1 (figures 8.5 and 8.6). Additional work (beyond the benchmark) is needed by MTF to gain

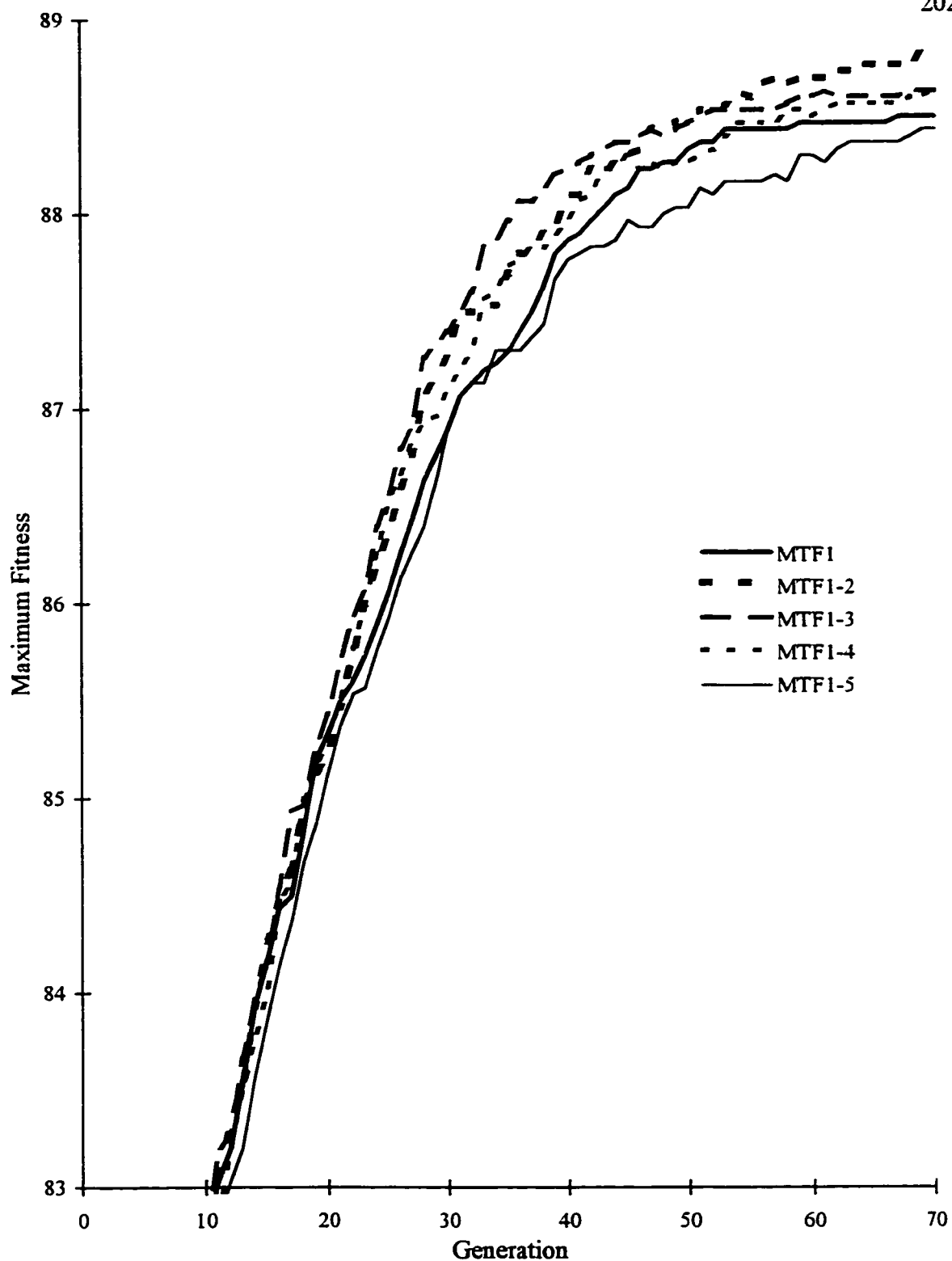


Figure 8.7: Comparison of MTF1 runs with MTF applied every i^{th} generation, $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of maximum fitness in each generation. Population size 64K.

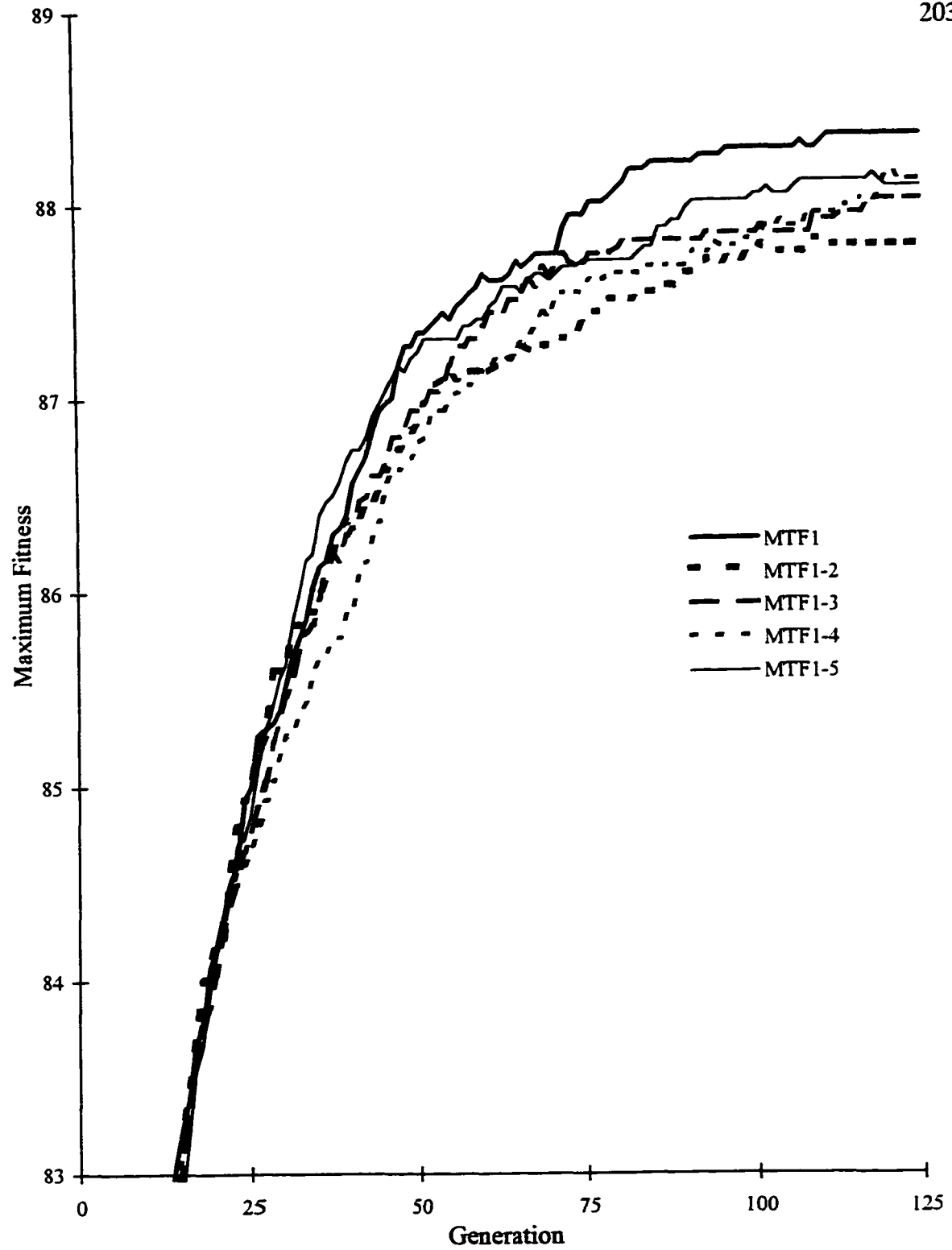


Figure 8.8: Comparison of MTF1 runs with MTF applied every i^{th} generation, $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of maximum fitness in each generation. Population size 16K.

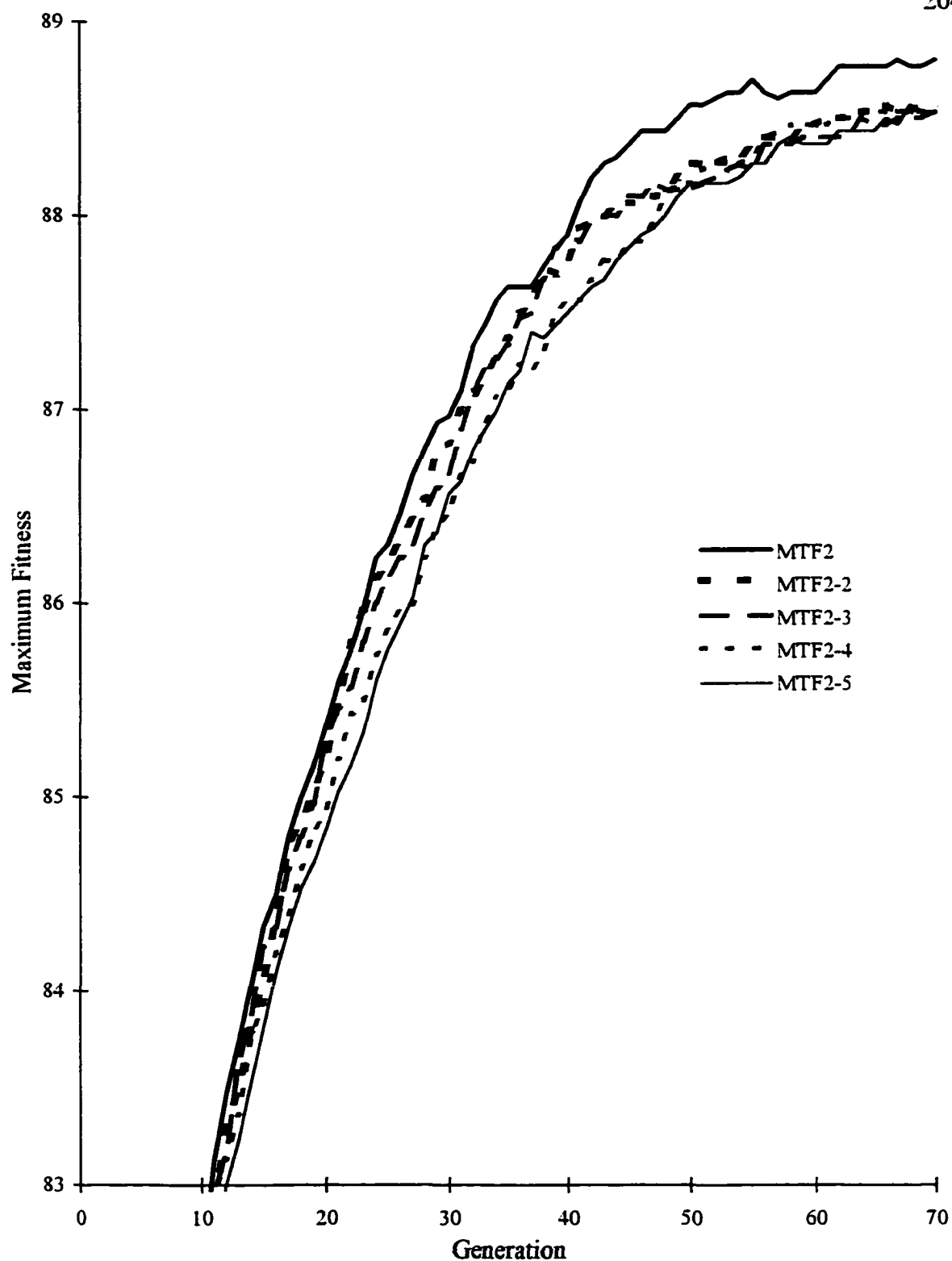


Figure 8.9: Comparison of MTF2 runs with MTF applied every i^{th} generation, $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of maximum fitness in each generation. Population size 64K.

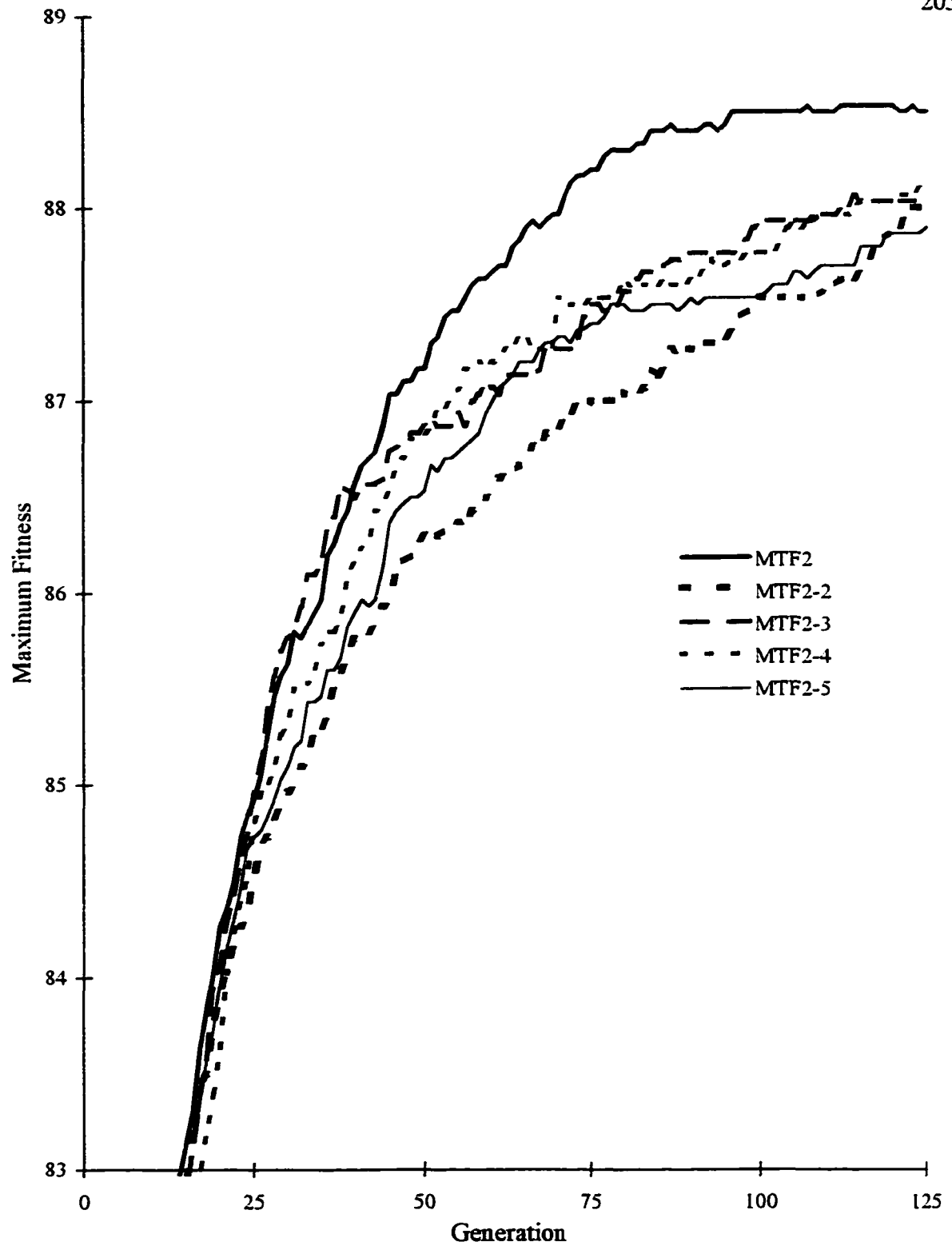


Figure 8.10: Comparison of MTF2 runs with MTF applied every i^{th} generation, $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of maximum fitness in each generation. Population size 16K.

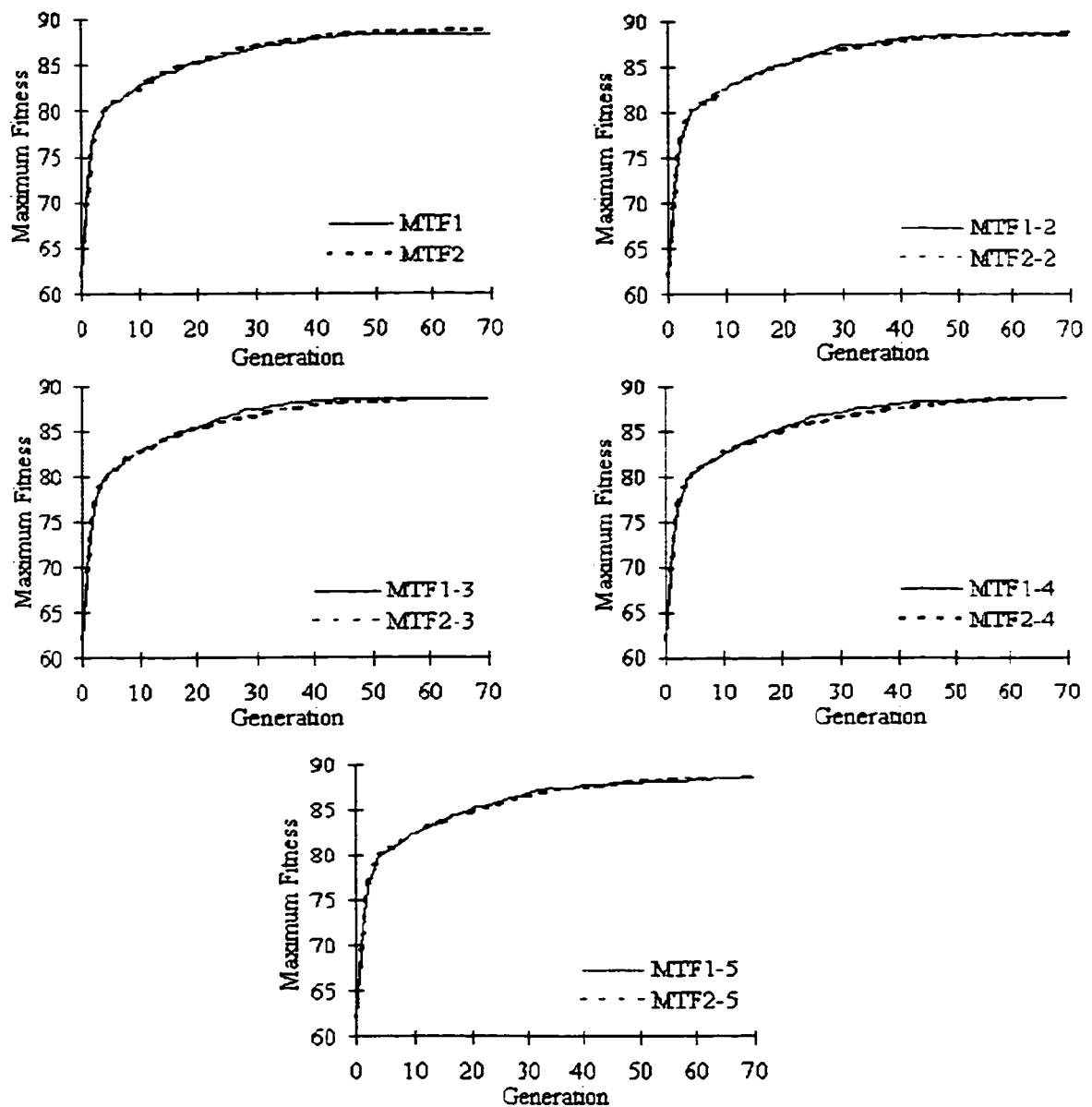


Figure 8.11: Comparison of MTF1- i and MTF2- i with MTF applied every i^{th} generation. $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of the maximum fitness in each generation. Population size 64K.

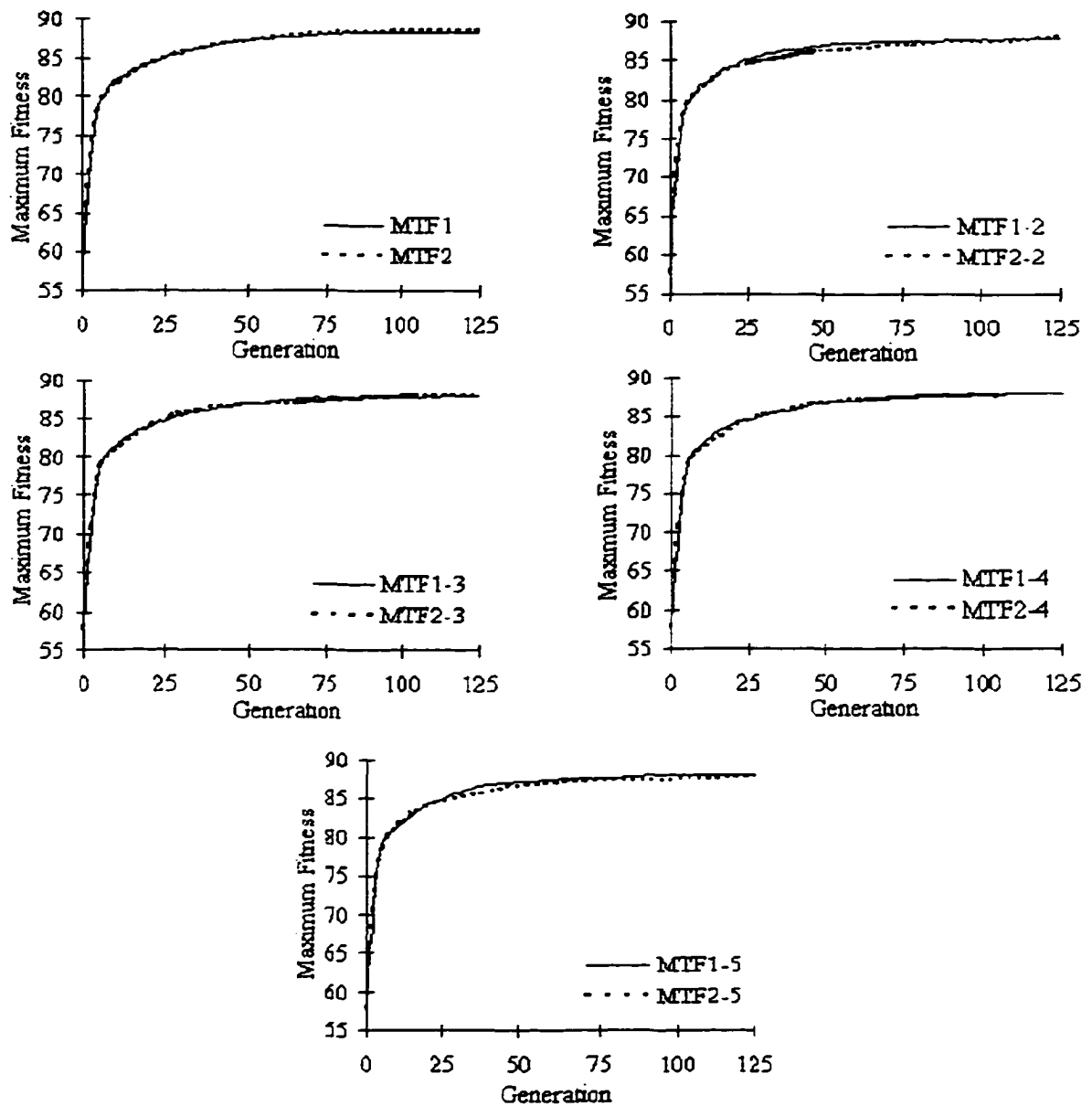


Figure 8.12: Comparison of MTF1- i and MTF2- i with MTF applied every i^{th} generation. $i \in \{1, 2, 3, 4, 5\}$. Averages over 30 runs of the maximum fitness in each generation. Population size 16K.

the performance advantage indicated in this chapter. Whether this advantage is lost in terms of processor time due to the additional processing will be discussed in section 8.2.2.

8.2 Evaluation of efficiency

The evaluation of the evolutionary process in the previous section indicates that MTF gives the GA a boost, but additional consideration must be given to the efficiency in terms of convergence to a solution. Section 8.2.1 describes the experiments which produced the data to permit the comparison of how the different methods and variants affect convergence/efficiency. In sections 8.2.2 and 8.2.3 the data produced by these experiments are examined.

8.2.1 The experimentation

This second round of experiments focussed on the outcome of each run. The experiments are labelled B (benchmark), M (MTF), or S (SFS). This letter is followed by a "1" when competition has not been implemented, or by a "2" indicating that competition has been included. When a method name contains "-i", $i \in \{2,3,4,5\}$, MTF was applied to generation 0, and then to every i^{th} generation after that. In all other cases, the additional methods incorporated into B1 are applied to every generation.

These experiments include runs with population sizes of 64K, 32K, and 16K FSMs, and a run was halted when a solution (perfect scoring ant) was found. If no

perfect scoring individual was located, the run was halted when it had bred generation *maxgen*. The value of *maxgen* depends on population size:

70 generations for a population of 64K FSMs,
 100 generations for a population of 32K FSMs, and
 125 generations for a population of 16kKFSMs.

To gauge the sensitivity of the different methods to *maxgen*, data were also collected for smaller values of *maxgen*:

50 and 60 generations for a population of 64K FSMs,
 75 generations for a population of 32K FSMs, and
 100 generations for a population of 16K FSMs.

The data for each generation were not recorded; only the outcome of each run was retained (generation in which the first solution(s) was/were found, processor time, number of solutions, and smallest number of states in a solution.)

Two sets of runs were carried out for this second type of experiment. One used the same 30 seeds that were used for the first type of experiment, those which were used to evaluate the evolutionary process. (See section 8.1.1 for details of this experimentation.) The second set of runs for each population size used 100 seeds and did not include any of the 30 seeds used in the first set of runs. The set of 30 seeds appears in section 4.3. The combination of three population sizes (64K, 32K, 16K), each with two distinct sets of seeds (30, 100) yield a total of six experiments. Timing data are available for only one of the experiments (16K population, 30 seeds).

The runs from the original 30 seeds are treated as separate experiments because of the vast amount of additional data that were collected in the first type of experiment for populations sizes 64K and 16K. It will be interesting to see if the

evaluation based on evolutionary progress applied to this set of runs (section 8.1.2) yields the same conclusions as the evaluation based on the outcome of each run (sections 8.2.2 and 8.2.3). The conclusions combining the results of the two different types of experiments appear in section 8.3.

8.2.2 Evaluation of the individual experiments

In this section the data on convergence are viewed in two ways. First in section 8.2.2.1 the data for each experiment are evaluated. Then in section 8.2.2.2 the same data are analyzed across the six experiments for each selected criterion. In figures 8.13 through 8.65, when a line in a table contains only a method name, the data are identical to the data on the previous line except when the figure contains rankings based on the **probability** that a method will get a solution first (figures 8.14, 8.15, 8.27, 8.35, 8.43, 8.51, and 8.59). In these seven figures the first column of numbers after the method name compares the method to the one on the line directly below. Consequently, the last line of each of these tables will contain only the method name. The purpose of omitting data due to replication is to emphasize the fact that the method on a line missing data has the identical ranking as the method immediately above it on the given list.

In the next six subsections (8.2.2.1 through 8.2.2.6), the data for each of the six experiments (16K, 32K, 64K population sizes; 30, 100 seeds) are evaluated. For each experiment rankings are provided based on different criteria. A summary for each of the experiments (sections 8.2.2.1 through 8.2.2.6) appears in the last

subsection of each section.

8.2.2.1 Population size 16K; 30 seeds

This experiment (first of the six) is especially important because it is the only experiment for which timing data are available. Based on this experiment, it is concluded that the processor time does not greatly affect the rankings. The SFS algorithm does not perform better than the benchmark and is frequently less efficient. Across all the different criteria, M1 and M2 perform better than the benchmark with varying degrees of improvement as measured by speedup or percent increase over B1. M2 is frequently more efficient than M1.

<i>maxgen</i> = 125				<i>maxgen</i> = 100			
M2	6	2.86	2.83	M2	8	3.48	3.26
M1	10	4.19	1.93	M1	12	5.02	2.26
M1-5	12	5.03	1.61	M1-5	13	5.51	2.06
M2-3	14	6.04	1.34	M1-3	15	6.64	1.71
M1-3				M2-3	16	7.33	1.55
M1-4				M1-4	17	8.11	1.40
M2-5	15	6.64	1.22	M1-2	18	9.02	1.26
B2				M2-5	19	10.08	1.13
M2-2	16	7.33	1.11	B2			
M1-2				B1	20	11.36	
M2-4				M2-2			
B1	17	8.11		M2-4	22	1.48	
S2	19	10.08		S2	24	2.06	
S1	22	14.84		S1	25	2.53	

Figure 8.13: Population size=16K; 30 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.1.1 Failure rate and K

Based on the number of failures and K (data in figure 8.13, description in

section 7.2.2.10), the SFS methods perform worse than the benchmark regardless of the value of *maxgen*. M1 and M2 topped both lists with speedups two to three times that of the benchmark. MTF with competition (M2) far surpasses M1's performance. When the MTFs are applied every i^{th} generation, $i \in \{2,3,4,5\}$, not all of them perform better than the benchmarks, and some are only marginally better than B1 (last column of each list of figure 8.13).

8.2.2.1.2 Probability of fewer generations or less processor time

Based on the probability that one method will get a solution in fewer generations than the next (data in next figure 8.14, description in section 7.2.2.10), S1 and S2 are the poorest performers appearing at the bottom of the lists in figure 8.14. M2 is most likely to get a solution first. M1 is also near the top of the lists, but nowhere nearly as impressive as M2. Note that M1-5 is ranked above M1 based on the first column after the method name, but when compared to the benchmark (last column of each list) it does not do as well as M1. This is determined by the percent increase of the probability (as computed by the program in figure 7.2 of section 7.3.10.2) of one method over another. The percent increase is based on probabilities. These probabilities are calculated by implementing a comparison of all pairs consisting of one value for each of two methods. Each value in the pair combines a number of generations which it took the GA to find a solution in a single successful run with a possible number of generations due to the failed runs preceding the successful run. For example, according to the data in figure 8.14, for

$maxgen=125$ M1-5 is more likely to get a solution sooner than M1 7.62% of the time.

The 7.62 is a percent increase, therefore the probability that M1-5 will get a solution before M1 is $100\% + 7.62\% \approx 108\%$ the probability for M1. When each is compared to B1 for $maxgen=125$, M1 is 192% better while M1-5 is only 163% better.

$maxgen = 125$			$maxgen = 100$		
M2	11.48%	353%	M2	6.39%	398%
M1-5	7.62	163	M1-5	12.81	218
M1	43.08	192	M1	37.90	229
M1-3	2.94	91	M1-3	7.60	127
M2-3	9.71	74	M2-3	16.85	95
M1-4	8.32	67	M1-4	1.57	77
M1-2	9.17	42	M1-2	22.78	58
M2-5	16.35	39	M2-5	23.56	41
B2	2.38	20	B2	16.45	16
M2-4	16.38	16	B1	11.46	
B1	5.19		M2-2	2.77	
M2-2	38.41		M2-4	76.27	
S2	33.86		S2	17.18	
S1			S1		

Figure 8.14: Population size=16K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is below B1.

When applicable, the *U Test* (section 7.3.10.3) generally yields degrees of confidence in excess of .95 indicating that the difference in performance of the selected methods with respect to the benchmark are indeed significant. The *U Test* support for the conclusions in the previous paragraph typify this generality. To obtain a degree of confidence for the conclusions based on figure 8.14, the *U Test* was applied to the lists of the number of generations needed to get a solution. These are the numbers which were used in the determination of the percent increases. Each method was compared to B1. When $maxgen = 125$, there is a .998 degree of

confidence that the difference in performance between each of the top six methods on the list (figure 8.14) and B1 is not due to chance; that is the difference in performance between each of the top six methods on the list and B1 is significant. The z scores based on the *U test* for these six methods range from a low of 3.51 for M1-4 to a high of 10.99 for M2 with the pattern of the z scores mirroring the pattern of the percent increases in figure 8.14. When *maxgen* = 100 the top five methods on the list are also within the .998 degree of confidence with the z score pattern with respect to B1 again following the pattern of the percent increases over B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	9.89%	284%	M2	4.75%	312%
M1-5	10.34	146	M1-5	16.03	193
M1	39.05	167	M1	34.05	197
M1-3	.68	77	M1-3	5.56	109
M2-3	11.26	61	M2-3	18.79	81
M1-4	10.63	53	M1-4	4.21	61
M1-2	6.58	32	M1-2	19.70	47
M2-5	13.68	33	M2-5	19.96	35
B2	5.39	19	B2	15.42	15
M2-4	12.46	12	B1	16.46	
B1	10.27		M2-2	2.26	
M2-2	42.94		M2-4	79.14	
S2	31.34		S2	13.19	
S1			S1		

Figure 8.15: Population size=16K; 30 seeds. Ranked using the probability that a method will get a solution first; based on processor time for a successful run combined with the processor time needed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is below B1.

There is a significant drop in performance for those methods appearing after M1 on both lists. The timing data, corresponding to figure 8.14 (description in section 7.2.2.10), are in figure 8.15. The rank order parallels that of figure 8.14, but the percentages are somewhat lower. This is because MTF, SFS, and competition

do more work than the benchmark for each individual. It had been surmised that by applying MTF every i^{th} generation, $i \in \{2,3,4,5\}$, rather than every generation, the processor time needed for a run might be reduced. However, based on the data presented in figures 8.14 and 8.15, the time saved by doing less work does not change the ranking. When the *U Test* is applied and $\text{maxgen} = 125$, the top three methods on the list (figure 8.15) fall into the .998 degree of confidence level with respect to B1. For $\text{maxgen} = 100$, the top five methods on the list have that same high degree of confidence that the difference in performance between each method and B1 is significant.

8.2.2.1.3 Grouping of best five runs

In figure 8.16, the best five runs based on the number of generations to a solution are grouped and compared. As mentioned earlier, this criterion for ranking is a modification of the work done by Angeline and Pollack (1993). M1 and M2 are near the top of the lists with speedups over B1 exceeding 2.0 on the **Fourth** and **Fifth Best** runs.

The timing data corresponding to figure 8.16 appear in figure 8.17. Here again M1 and M2 appear near the top of the lists with speedups somewhat less than in figure 8.16 due to the additional work the MTFs and competition do on each individual. The effect of the additional work on processor time is particularly evident in certain cases. In all of the cases, the processor time for a run (figure 8.17) changes the rank order only when the performance of methods is relatively close as

seen in figure 8.16. For example, compare **Fifth best** S2 and B1, the last three methods on the **Fourth best** lists, and **Second best** M1-3 through B1 of figure 8.16. In these cases, methods which are close in figure 8.16 are also close in figure 8.17, but the order is altered with those doing less work having slightly lower processor times.

Best runs			Second best			Third best			Fourth best			Fifth best		
M2	28	1.64	M1	34	1.38	M1-5	36	1.83	M1-5	38	2.24	M1-5	39	2.23
M1	30	1.53	M1-2	35	1.34	M2	40	1.65	M1	41	2.07	M1	43	2.02
M1-2	31	1.48	M1-5	36	1.31	M1			M2	43	1.98	M2	45	1.93
M1-5	34	1.35	M2	38	1.24	M2-3	42	1.57	M2-3	45	1.89	M2-3	46	1.89
M2-3	36	1.28	M2-3			M1-3	45	1.47	M1-3	47	1.81	M1-3	48	1.81
M2-5	37	1.24	M1-3	45	1.04	M1-2	46	1.43	M1-2	48	1.77	M1-2	49	1.78
M1-3	40	1.15	M1-4			M1-4			M1-4	49	1.73	M1-4	55	1.58
M2-4			M2-5	46	1.02	M2-5	51	1.29	M2-4	57	1.49	M2-4	57	1.53
M1-4	41	1.12	B1	47		M2-4	53	1.25	M2-5	60	1.42	M2-5	61	1.43
B1	46		M2-4	50		B2	62	1.06	B2	68	1.25	B2	69	1.26
M2-2	47		S1	50		B1	66		M2-2	71	1.20	S2	85	1.02
S1	50		B2	53		M2-2	68		S2	81	1.05	B1	87	
B2	50		M2-2	62		S1	77		S1	83	1.02	M2-2	87	
S2	68		S2	69		S2	78		B1	85		S1	89	

Figure 8.16: Population size=16K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.).

Best runs			Second best runs			Third best runs		
M2	503.17	1.55	M1	612.86	1.30	M1-5	639.81	1.75
M1	539.71	1.45	M1-2	624.45	1.28	M2	716.15	1.56
M1-2	554.41	1.41	M1-5	638.28	1.25	M1	720.64	1.55
M1-5	603.72	1.29	M2-3	668.66	1.19	M2-3	738.36	1.52
M2-3	634.66	1.23	M2	681.01	1.17	M1-3	799.53	1.40
M2-5	648.22	1.20	M1-4	797.21	1.00	M1-4	815.90	1.37
M2-4	701.36	1.11	B1	797.84		M1-2	819.73	1.37
M1-3	710.97	1.10	M1-3	798.89		M2-5	891.48	1.26
M1-4	728.11	1.07	M2-5	804.73		M2-4	928.55	1.21
B1	780.96		M2-4	878.03		B2	1071.32	1.04
M2-2	827.04		S1	883.45		B1	1118.95	
B2	862.60		B2	914.99		M2-2	1199.26	
S1	883.09		M2-2	1092.69		S1	1356.41	
S2	1217.43		S2	1235.09		S2	1391.99	

Figure 8.17: Population size=16K; 30 seeds. Ranked by the processor time taken to find a solution—grouped runs. The first number following the method name is the processor time. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. Continued on next page.

Fourth best runs			Fifth best runs		
M1-5	675.03	2.13	M1-5	691.60	2.13
M1	738.08	1.95	M1	774.14	1.90
M2	767.93	1.87	M2	803.85	1.83
M2-3	792.33	1.82	M2-3	808.39	1.82
M1-3	834.42	1.72	M1-3	852.67	1.73
M1-2	854.62	1.68	M1-2	871.25	1.69
M1-4	870.05	1.65	M1-4	973.29	1.51
M2-4	999.44	1.44	M2-4	1000.52	1.47
M2-5	1047.46	1.37	M2-5	1063.78	1.38
B2	1171.17	1.23	B2	1188.42	1.24
M2-2	1250.58	1.15	B1	1471.30	
B1	1438.28		S2	1521.62	
S2	1449.49		M2-2	1528.96	
S1	1465.74		S1	1568.37	

Figure 8.17 continued: Population size=16K; 30 seeds. Ranked by the processor time taken to find a solution—grouped runs. The first number following the method name is the processor time. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.1.4 G_{AV} and $T_{AV:S}$

In figure 8.18, G_{AV} (description in section 7.2.1) is the basis for comparison. and in figure 8.19 the basis for ranking is the corresponding processor times. The performance of M1 and M2 are close, both having small speedups over B1. M1 is minimally better than M2 on all four lists. M1-5 tops these four lists and, looking back over figures 8.13 through 8.17, it is also near the top of all the previous lists. The benchmarks and SFSs appear towards the bottom of all four lists in figures 8.18 and 8.19, with no consistency of order.

When the *U Test* is applied to the values used to calculate G_{AV} and $maxgen = 125$, M1-5 has a .998 degree of confidence with respect to B1; M1 and M2 have a .99 degree of confidence. At $maxgen = 100$, M1-5 has a .99 degree of confidence, and M1 and M2 have a .98 degree of confidence. For both values of $maxgen$ and $T_{AV:S}$, M1-5 has a .99 degree of confidence and M1 has a .98 degree of confidence. M2 has a .97 degree of confidence for $maxgen = 125$, and .95 for $maxgen = 100$. For

both G_{AV} and $T_{AV:S}$, the *U Test* indicates that the difference in the performance between each of the methods of interest and the benchmark is significant.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M1-5	55.78	1.55	M1-5	52.12	1.52
M1	64.20	1.35	M1-2	57.92	1.36
M1-3	64.56	1.34	M2-4	58.75	1.34
M2	65.21	1.33	M1	59.06	1.34
M1-2			M2	61.18	1.29
M2-3	68.88	1.26	M1-3	61.33	1.29
M1-4	72.12	1.20	M2-3	62.57	1.26
M2-5	77.20	1.12	M2-5	62.73	1.26
M2-4	80.57	1.08	M1-4	63.62	1.24
S1	85.38	1.01	S1	69.80	1.13
B2	85.47	1.01	B2	75.36	1.05
B1	86.62		S2	77.83	1.02
M2-2	92.57		B1	79.00	
S2	95.82		M2-2	81.90	

Figure 8.18: Population size = 16K; 30 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M1-5	986.40	1.49	M1-5	922.26	1.45
M1-3	1143.83	1.28	M2-4	1029.19	1.30
M1	1151.29	1.27	M1-2	1030.58	1.30
M1-2	1159.56	1.26	M1	1059.48	1.26
M2	1162.47	1.26	M1-3	1086.67	1.23
M2-3	1208.58	1.21	M2	1091.01	1.23
M1-4	1275.80	1.15	M2-5	1094.88	1.22
M2-5	1345.40	1.09	M2-3	1098.56	1.22
M2-4	1409.86	1.04	M1-4	1126.02	1.19
B1	1466.42		S1	1231.41	1.09
B2	1472.39		B2	1298.82	1.03
S1	1504.07		B1	1337.99	
M2-2	1627.73		S2	1392.09	
S2	1711.44		M2-2	1441.04	

Figure 8.19: Population size = 16K; 30 seeds. Ranked by $T_{AV:S}$. The first number following the method name is $T_{AV:S}$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.1.5 $G\mu$ and $T\mu$

In figure 8.20, $G\mu$ is calculated using the formula to predict values for the population of all seeds (section 7.2.2.5), and in figure 8.21 the corresponding $T\mu$ is used (section 7.2.2.7). Figures 8.22 and 8.23 contain the data for $G\mu$ and $T\mu$ respectively, using the formulas which calculate these values for the sample (section 7.2.2.9). The only time the rank order differs for $G\mu$ and the corresponding $T\mu$ is when performance is very close. As before, speedups based on $T\mu$ are smaller than for $G\mu$. M2, M1, and M1-5 appear at the top of the eight lists in these four figures. The speedups over B1 for the three MTF methods at the top of the lists range from a low of 1.70 (figure 8.23, *maxgen*=125, M1-5) to a high of 2.85 (figure 8.20, *maxgen*=100, M2). S2 and S1 are at the bottom of all these lists, generally performing far worse than B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	97.71	2.58	M2	98.91	2.85
M1	128.20	1.97	M1	127.39	2.21
M1-5	140.78	1.79	M1-5	130.36	2.16
M1-3	175.81	1.44	M1-3	163.33	1.73
M2-3	180.13	1.40	M2-3	179.00	1.58
M1-4	183.37	1.38	M1-4	196.70	1.43
M2-5	204.20	1.24	M1-2	210.42	1.34
M1-2	210.21	1.20	M2-5	238.18	1.18
B2	212.47	1.19	B2	250.81	1.12
M2-4	225.57	1.12	B1	282.00	
M2-2	237.57	1.06	M2-2	284.90	
B1	252.39		M2-4	337.50	
S2	314.46		S2	482.83	
S1	432.88		S1	575.80	

Figure 8.20: Population size=16K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	1718.11	2.46	M2	1738.19	2.72
M1	2268.64	1.87	M1	2252.39	2.10
M1-5	2455.14	1.72	M1-5	2270.80	2.08
M1-3	3076.04	1.38	M1-3	2854.87	1.65
M2-3	3123.88	1.36	M2-3	3101.17	1.53
M1-4	3206.08	1.32	M1-4	3435.88	1.37
M2-5	3520.26	1.20	M1-2	3692.67	1.28
B2	3623.49	1.17	M2-5	4101.92	1.15
M1-2	3693.00	1.15	B2	4272.87	1.11
M2-4	3906.17	1.08	B1	4724.05	
M2-2	4136.86	1.02	M2-2	4956.86	
B1	4232.87		M2-4	5838.55	
S2	5563.10		S2	8533.85	
S1	7546.20		S1	10026.76	

Figure 8.21: Population size=16K; 30 seeds. Ranked by $T\mu$ calculated using the formula for the population. The first number following the method name is $T\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	96.45	2.49	M2	97.31	2.71
M1	125.20	1.92	M1	123.85	2.13
M1-5	136.36	1.76	M1-5	126.06	2.09
M1-3	169.32	1.42	M1-3	157.02	1.68
M2-3	173.64	1.39	M2-3	171.30	1.54
M1-4	176.88	1.36	M1-4	187.26	1.41
M2-5	196.32	1.25	M1-2	198.77	1.33
M1-2	200.61	1.20	M2-5	223.65	1.18
B2	204.60	1.18	B2	236.28	1.12
M2-4	215.97	1.11	B1	263.64	
M2-2	227.97	1.06	M2-2	266.54	
B1	240.62		M2-4	306.64	
S2	296.32		S2	425.12	
S1	394.38		S1	491.63	

Figure 8.22: Population size=16K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	1695.88	2.38	M2	1710.05	2.58
M1	2215.43	1.82	M1	2189.61	2.02
M1-5	2377.84	1.70	M1-5	2195.88	2.01
M1-3	2962.38	1.36	M1-3	2744.36	1.61
M2-3	3011.21	1.34	M2-3	2967.66	1.49

Figure 8.23: Population size=16K; 30 seeds. Ranked by $T\mu$ calculated using the formula for the sample. The first number following the method name is $T\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. **Continued on next page.**

<i>maxgen</i> = 125 (continued)			<i>maxgen</i> = 100 (continued)		
M1-4	3092.53	1.30	M1-4	3270.89	1.35
M2-5	3384.33	1.19	M1-2	3487.90	1.27
B2	3489.05	1.16	M2-5	3851.34	1.15
M1-2	3524.10	1.15	B2	4025.03	1.10
M2-4	3739.75	1.08	B1	4416.23	
M2-2	3969.59	1.02	M2-2	4637.24	
B1	4035.27		M2-4	5304.18	
S2	5242.13		S2	7513.60	
S1	6874.86		S1	8560.87	

Figure 8.23 *continued*: Population size=16K; 30 seeds. Ranked by $T\mu$ calculated using the formula for the sample. The first number following the method name is $T\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

The *U Test* can only be used to obtain a degree of confidence for the calculations for the sample because the *U Test* requires finite sets. In figure 8.22, for *maxgen* = 125 the top six methods on the list fall within the .998 degree of confidence and for *maxgen* = 100 the top five have the same high degree of confidence. In figure 8.23 the top three methods for *maxgen* = 125 and the top five for *maxgen* = 100 also have a .998 degree of confidence. Thus, the difference in the performance between each of these methods and B1 is significant.

8.2.2.1.6 Machine size

With respect to machine size, the grouped runs are presented in figure 8.24. The better runs indicate only small differences between the methods. Stratification occurs as the groupings move away from the **Best runs**. With a *maxgen* of 125 in the **Second best**, B1 and S1 start moving away from the methods which produce smaller successful FSMs. By the **Third best**, the benchmarks and SFSs are separated from the closely bunched MTFs. As the data progress from **Best runs**

maxgen = 125

Best runs	Second best	Third best	Fourth best
M2-5 8 11%	M2 9 31%	M1 9 36%	M2-4 9 53%
All other	M1	M2-3	M2-5
methods 9	M2-3	M2-4	M2 10 47%
	M1-3	M2-5	M1
	M2-4	M1-5	M1-2
	M1-4	M2 10 29%	M1-3
	M2-5	M2-2	M1-4
	M1-5	M1-2	M1-5
	B2	M1-3	M2-2 11 42%
	M2-2 10 23%	M1-4	M2-3
	M1-2	S2 12 14%	S2 14 26%
	S2	B1 14	B2 15 21%
	B1 13	B2 15	B1 19
	S1	S1 16	S1
Fifth best	Sixth best	Seventh best	Eighth (last)
M1 10 52%	M2-4 10 57%	M2 11 52%	M1 11 54%
M1-3	M2-5	M1	M1-3
M2-4	M1-5	M2-3	M1-4
M1-4	M2 11 52%	M1-3	M1-5
M2-5	M1	M1-4	M2 12 50%
M1-5	M1-2	M2-5	M2-3
M2 11 48%	M2-3	M1-5	M2-5
M2-2	M1-3	M2-2 12 48%	M2-2 13 46%
M1-2	M1-4	M1-2	M1-2
M2-3	M2-2 12 48%	M2-4	M2-4 14 42%
B2 15 29%	S2 16 30%	B2 16 30%	B2 17 29%
S2 16 24%	B2	S2 19 17%	S2 20 17%
B1 21	B1 23	B1 23	B1 24
S1	S1	S1	S1 27

maxgen = 100

Best runs	Second best	Third best	Fourth best
M2-5 8 11%	M2 9 36%	M1 9 61%	M2-4 9 61%
All methods	M1	M2-3	M2-5
not listed 9	M2-3	M2-4	M2 10 57%
S2 10	M1-3	M2-5	M1
	M2-4	M1-5	M1-2
	M1-4	M2 10 57%	M1-3
	M2-5	M1-2	M1-4
	M1-5	M1-3	M1-5
	M1-2 10 29%	M1-4	M2-3 11 52%
	M2-2 11 21%	M2-2 11 52%	M2-2 12 48%
	S2 12 14%	S2 14 39%	B2 16 30%
	B1 14	B2 15 35%	S2 19 17%
	B2 15	S1 21 9%	B1 23
	S1 16	B1 23	S1

Figure 8.24: Population size=16K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method names is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.) . Continued on next page.

maxgen = 100 (continued)

Fifth (last)		
M1-3, M1-4, M2-5, M1-5	10	58%
M2, M1, M1-2, M2-3	11	54%
M2-2	12	50%
M2-4, B2	16	33%
S2	20	17%
S1	23	4%
B1	24	

Figure 8.24 **continued**: Population size=16K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method names is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M1-5	11.67	44.82%	M2-5	11.18	49.64%
M2-5	12.07	42.93	M1-5	11.59	47.79
M1-4	12.19	42.36	M1-4	11.69	47.34
M1-3	12.31	41.80	M1-3	11.80	46.85
M1-2	12.57	40.57	M1-2	12.58	43.33
M1	12.65	40.19	M1	12.89	41.94
M2-2	13.21	37.54	M2-3	13.57	38.87
M2-3	13.38	36.74	M2-2	13.70	38.29
M2	13.75	34.99	M2	13.91	37.34
M2-4	14.50	31.44	M2-4	14.88	32.97
B2	16.60	21.51	S2	16.67	24.91
S2	17.45	17.49	B2	17.18	22.61
S1	18.88	10.73	S1	18.40	17.12
B1	21.15		B1	22.20	

Figure 8.25: Population size=16K; 30 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1.

to **Eighth (last)**, the MTFs spread out a bit, but the clear gap between the MTFs and the other methods remains. When *maxgen*≈100, the gap between the MTFs and the other methods again appears in the **Third best**, but M2-4 drops down to the lower levels on the **Fifth (last)** list. When the average machine size is considered (figure 8.25), the MTFs produce the smallest machines and B1 the largest. When the *U Test* is applied and *maxgen*=125 all the MTFs, except M2-4, are within the .998

degree of confidence indicating a significant difference in the performance when compared to B1. When *maxgen*=100, the confidence level for the MTFs, again except for M2-4, is .99 with some remaining at the .998 level.

8.2.2.1.7 Summary of experiment results

Based on the data presented in the previous subsections, M1 and M2, while not always at the top, are consistently better than the benchmarks and SFSs. M1-5 also fares well. Looking over the data which take into account the failure rate and at the three top MTFs (M1, M2, M1-5) in figures 8.13-8.15 and 8.20-8.23, based on speedup and percent increase over B1 performance improves with decreasing *maxgen*. When the *U Test* is applicable, there is a high degree of confidence that the difference in performance between each of the selected methods and B1 is significant. The superiority of M1 and M2 is also evident when looking at the evolutionary progress (figure 8.6) corresponding to this experiment.

The processor time is not available for any of the remaining experiments, but the data from this experiment indicate that the ranking based on processor time sometimes differs from the ranking induced by the corresponding number of generations, but only when the performance of methods is close. When this occurs, the performance of the affected methods is also very close based on processor time even though the rank order may be slightly different. The *U Test* indicates that the performance of the highly ranked methods based on processor time is significant when compared to B1, and the degrees of confidence based on processor time are

minimally lower than those obtained when the basis of comparison is the number of generations. The next experiment differs from this one in the population size, and values of *maxgen*.

8.2.2.2 Population size 32K; 30 seeds

For this experiment the population size is double that of the previous experiment. With more individuals being examined during each generation, fewer generations are generally needed. Consequently, the values for *maxgen* were decreased. Nothing else was changed from the previous experiment. The timing data were recorded, but it is not considered usable due to differences in the processors on which these runs were executed. With one exception, M1 and M2 outperformed the benchmarks and SFSs, and M1's performance far surpasses that of M2.

8.2.2.2.1 Failure rate and K

Based on the number of failures and K (data in figure 8.26, description in section 7.2.2.10), the SFS methods perform worse than the benchmark regardless of the value of *maxgen*. M1 performs fair to good with speedups of 1.58 and 2.33. M2 shows no advantage over B1 with *maxgen*=100. When the MTFs are applied every i^{th} generation, $i \in \{2,3,4,5\}$, all perform better than the benchmark, but some are only marginally better than B1 (last column of each list of figure 8.26).

<i>maxgen</i> = 100				<i>maxgen</i> = 75			
M1-3	4	2.29	2.41	M1-4	6	2.86	2.83
M1-2	5	2.57	2.14	M1-3	7	3.16	2.56
M1-4				M1	8	3.48	2.33
M2-3	6	2.86	1.92	M1-2	10	4.19	1.93
M1	8	3.48	1.58	M2-3	11	4.59	1.77
M2-4	9	3.82	1.44	M2-4			
M1-5				M2-5	12	5.03	1.61
M2-5	10	4.19	1.31	M2	14	6.04	1.34
M2-2	11	4.59	1.20	M2-2			
B2	12	5.03	1.10	M1-5	16	7.33	1.11
B1	13	5.51		B1	17	8.11	
M2				B2	19	10.08	
S2	16	7.32		S2	20	11.36	
S1	18	9.02		S1	21	12.91	

Figure 8.26: Population size=32K; 30 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1	17.92%	345%	M1	20.42%	472%
M1-4	27.35	370	M1-4	31.32	438
M1-3	5.47	248	M1-3	8.67	266
M1-2	12.06	198	M1-2	4.98	188
M2-3	5.35	199	M2-5	2.72	136
M2-5	5.43	110	M2-3	9.95	207
M2-4	21.12	122	M2-4	16.10	146
M2	.94	62	M2	10.35	95
M1-5	5.41	75	M2-2	5.72	70
M2-2	50.47	60	M1-5	64.96	65
B2	6.15	6	B1	9.32	
B1	12.22		B2	1.70	
S2	20.37		S2	16.49	
S1			S1		

Figure 8.27: Population size=32K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is below B1.

8.2.2.2.2 Probability of fewer generations

Based on the probability that one method will get a solution in fewer

generations than the next (data in figure 8.27, description in section 7.2.2.10), S1 and S2 are the poorest performers appearing at the bottom of these lists. M1 is most likely to get a solution first. M2 is somewhat better than B1, but nowhere nearly as impressive as M1. M1-4, M1-3, and M2-3 also performed well with a 200% or better advantage over B1. The *U Test* indicates a .998 degree of confidence that the difference in performance with respect to B1 is significant for all the MTFs except M2 when *maxgen* = 100, and for all the MTFs except M1-5 when *maxgen* = 75. For M2 with *maxgen* = 100, the degree of confidence is at the very low level of .57.

Best runs	Second best	Third best	Fourth best	Fifth best
M1-4 25 2.16	M1-4 28 1.93	M1 31 1.77	M1 31 1.81	M1 36 1.58
M2-3 27 2.00	M2-3 29 1.86	M1-4	M2-5 34 1.65	M2-2 39 1.46
M1 28 1.93	M1 30 1.80	M2 33 1.67	M2-2 38 1.47	M1-4
M2-4 29 1.86	M2 33 1.64	M2-5	M1-2 39 1.44	M2-5
M2 31 1.74	M2-5	M2-3 36 1.53	M1-3	M1-2 41 1.39
M2-5 32 1.69	M1-2 35 1.54	M2-2 38 1.45	M2-4	M1-3
S2 33 1.64	M2-4 36 1.50	M1-2	M1-4	M2 42 1.36
M2-2 34 1.59	M2-2 37 1.46	M2-4	M2 42 1.33	M2-3
M1-2	M1-5	M1-3 39 1.41	M2-3	M2-4
S1 35 1.54	M1-3 39 1.38	S2 41 1.34	S2 44 1.27	S2 45 1.27
M1-5 37 1.46	S2	M1-5 44 1.25	M1-5 45 1.24	M1-5 46 1.24
M1-3 38 1.42	S1	S1 47 1.17	S1 52 1.08	B2 56 1.02
B2	B2 43 1.26	B2 54 1.02	B2 55 1.02	B1 57
B1 54	B1 54	B1 55	B1 56	S1

Figure 8.28: Population size=32K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1 (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.).

8.2.2.2.3 Grouping of best five runs based on number of generations

In figure 8.28, the best five runs based on the number of generations to a solution are grouped and compared. M1 is near the top of the lists with speedups

over B1 falling between 1.58 and 1.93. On all five lists the benchmarks are at the bottom and on all but the **Best runs** there is a gap, sometimes a large one, between the benchmarks and the methods which appear above them on the lists.

8.2.2.2.4 G_{AV}

In figure 8.29, G_{AV} (description in section 7.2.1) is the basis for comparison. The performance of M1 and M2 are close, both having small speedups over B1. M1 tops both lists and M2 falls between M1 and B1. The *U Test* gives a .998 degree of confidence for M1 and a .99 degree of confidence for M2 across the two *maxgens* that the difference in performance with respect to B1 is significant. The SFSs tend to be only slightly better than the benchmark.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1	45.27	1.53	M1	45.27	1.37
M1-4	50.40	1.38	M2-5	48.78	1.27
M2-5	51.90	1.34	M1-4	49.08	1.27
M2	52.18	1.33	M2-3	49.42	1.26
M2-4	55.05	1.26	M2	49.44	1.26
M1-2	56.68	1.22	M1-2	50.85	1.22
M1-3	57.46	1.21	M1-5	51.29	1.21
M2-2	58.16	1.19	S2	51.50	1.21
M2-3	58.17	1.19	M2-4	52.21	1.19
S1	61.58	1.13	M1-3	53.09	1.17
S2	62.71	1.11	M2-2	53.19	1.17
M1-5	64.33	1.08	S1	53.44	1.16
B1	69.41		B2	57.82	1.07
B2	72.11		B1	62.15	

Figure 8.29: Population size=32K; 30 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1-4	71.60	2.06	M1-4	69.08	2.35
M1-3	74.00	2.00	M1	73.91	2.20
M1-2	77.88	1.90	M1-3	77.22	2.10
M1	83.00	1.78	M1-2	89.85	1.81
M2-3	84.42	1.75	M2-3	94.42	1.72
M2-4	99.34	1.49	M2-4	97.21	1.67
M2-5	103.40	1.43	M2-5	100.45	1.62
M1-5	108.62	1.36	M2	116.94	1.39
M2-2	117.63	1.26	M2-2	120.69	1.35
M2	130.42	1.13	M1-5	139.15	1.17
B2	140.44	1.05	B1	162.53	
B1	147.65		B2	190.09	
S2	179.14		S2	204.50	
S1	214.08		S1	231.77	

Figure 8.30: Population size=32K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1-4	70.82	2.02	M1-4	68.32	2.28
M1-3	73.42	1.95	M1	72.70	2.14
M1-2	77.10	1.86	M1-3	76.26	2.04
M1	81.40	1.76	M1-2	88.04	1.77
M2-3	83.41	1.72	M2-3	92.22	1.69
M2-4	97.37	1.47	M2-4	95.01	1.64
M2-5	101.00	1.42	M2-5	97.78	1.59
M1-5	106.65	1.34	M2	113.03	1.38
M2-2	114.71	1.25	M2-2	116.78	1.33
M2	126.12	1.14	M1-5	133.36	1.17
B2	136.90	1.05	B1	155.44	
B1	143.35		B2	179.15	
S2	171.44		S2	190.68	
S1	202.43		S1	214.04	

Figure 8.31: Population size=32K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.2.5 $G\mu$

In figure 8.30, $G\mu$ is calculated using the formula to predict the value for the population of all seeds (section 7.2.2.5), and figure 8.31 contains the data for $G\mu$ using the formula which calculates this value for the sample (section 7.2.2.9). For M1 the speedup over B1 has a low of 1.76 (figure 8.31, *maxgen*=100) and a high of 2.20

(figure 8.30, *maxgen*=75). M2 is also better than B1, but not as strong as M1. S2 and S1 are at the bottom of these lists, performing worse than B1, sometimes much worse. The *U Test* can be used for the data in figure 8.31. For M1 with *maxgen* = 100 . and M1 and M2 with *maxgen* = 75, the degree of confidence that the difference in performance with respect to B1 is significant is .998. M2 with *maxgen* = 100 falls short with only a .57 degree of confidence.

8.2.2.2.6 Machine size

With respect to machine size, the grouped runs are presented in figure 8.32. The better runs indicate only small differences between the methods. Stratification occurs as the groupings move away from the **Best runs**. With a *maxgen* of 100, in the **Third best**, S1 and S2 start moving away from the methods which produce smaller successful FSMs. By the **Fourth best**, the benchmarks and SFSs are separated from the closely bunched MTFs. As the data progress from **Best runs** to **Twelfth (last)**, the MTFs alternately spread out a bit and move closer, but the clear gap between the MTFs and the other methods remains. When *maxgen*=75, the gap between the MTFs and the other methods again appears in the **Fourth best**, and remains through the **Ninth (last)** list. When the average machine size is considered (figure 8.33), the MTFs produce the smallest machines with a clear separation between the MTFs and the other methods. For both values of *maxgen* the degree of confidence that the difference in performance of each of the MTFs with respect to B1 is not due to chance exceeds .98. M2-4 is at the .98 level of

confidence. All other MTFs exceed .99 with some falling within the .998 level.

maxgen = 100

Best runs			Second best			Third best			Fourth best		
M2	8	11%	M2	8	20%	M2	8	20%	M2	9	36%
M2-2			M1	9	10%	M1	9	10%	M1		
M2-3			M2-2			M2-2			M2-2		
M2-4			M1-2			M1-3			M1-3		
M2-5			M2-3			M1-4			M1-4		
M1-5			M1-3			M2-5			M1-2	10	29%
B1	9		M1-4			M1-5			M2-3		
M1			M2-5			B1	10		M2-4		
M1-2			M1-5			M1-2			M2-5		
M1-3			B1	10		M2-3			M1-5		
M1-4			M2-4			M2-4			B2	13	7%
S1	10		S1	11		B2	11		B1	14	
S2	11		B2			S1	13		S2	17	
B2			S2	14		S2	16		S1	18	
Fifth best			Sixth best			Seventh best			Eighth best		
M2	9	40%	M2-2	9	40%	M2	10	44%	M2	10	55%
M2-2			M1-3			M1			M1		
M1-3			M2	10	33%	M2-2			M1-3		
M1	10	33%	M1			M1-3			M1-5		
M1-2			M2-5			M2-5			M2-2	11, 50%	
M1-4			M1-5			M1-5			M1-2		
M2-5			M1-2	11	27%	M1-2	11	39%	M2-3		
M1-5			M2-3			M2-3			M2-4		
M2-3	11	27%	M2-4			M2-4			M1-4		
M2-4			M1-4			M1-4			M2-5		
B2	14	7%	B1	15		B1	18		B2	20	9%
B1	15		B2	17		B2			B1	22	
S2	18		S2	18		S2	20		S2		
S1	21		S1	22		S1	23		S1	24	
Ninth best			Tenth best			Eleventh best			Twelfth (last)		
M1	10	55%	M1	10	57%	M1	10	57%	M1-3	11	54%
M1-3			M1-3			M1-3	11	52%	M1-4		
M1-5			M2	11	52%	M1-4			M1-5		
M2	11	50%	M2-3			M1-5			M2	12	50%
M2-3			M1-4			M2	12	48%	M1		
M2-4			M2-5			M1-2			M1-2		
M1-4			M1-5			M2-3			M2-3		
M2-5			M2-2	12	48%	M2-5			M2-5		
M2-2	12	45%	M1-2			M2-2	13	43%	M2-2	13	46%
M1-2			M2-4			M2-4			M2-4		
B2	20	9%	B2	21	9%	B2	22	4%	B2	22	8%
B1	22		B1	23		B1	23		B1	24	
S2	23		S1	24		S2	26		S2	26	
S1	24		S2	26		S1	28		S1	30	

Figure 8.32: Population size=32K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.) Continued on next page.

maxgen = 75

Best runs			Second best			Third best			Fourth best		
M2	8	20%	M2	8	20%	M2	8	43%	M2	9	50%
M2-2			M1	9	10%	M1	9	36%	M1		
M2-3			M2-2			M2-2			M2-2		
M2-4			M2-3			M1-3			M1-3		
M2-5			M1-3			M1-4			M1-2	10	44%
M1	9	10%	M1-4			M2-5			M2-4		
M1-2			M2-5			M1-2	10	29%	M1-4		
M1-3			B1	10		M2-4			M2-5		
M1-4			M1-2			M1-5			M1-5		
M1-5			M2-4			M2-3	11	21%	M2-3	11	39%
B1	10		M1-5			B2			B2	13	28%
S1			S1	11		S1	13	7%	S2	17	6%
S2	11		B2			B1	14		B1	18	
B2			S2	14		S2	16		S1		
Fifth best			Sixth best			Seventh best			Eighth best		
M2	9	59%	M2-2	9	59%	M2	10	57%	M1	10	57%
M2-2			M1-3			M1			M1-3		
M1-3			M2	10	55%	M2-2			M2	11	52%
M1	10	55%	M1			M1-3			M2-2		
M2-5			M2-5			M1-2	11	52%	M2-4		
M1-5			M1-5			M2-4			M1-4		
M1-2	11	50%	M1-2	11	50%	M1-4			M2-5		
M2-3			M2-3			M2-5			M1-5		
M2-4			M2-4			M1-5			M1-2	12	48%
M1-4			M1-4			M2-3	12	48%	M2-3		
S2	18	18%	S2	18	18%	S2	20	13%	S2	22	4%
B2	20	9%	B2	20	9%	B2	21	9%	B2		
B1	22		B1	22		B1	23		B1	23	
S1			S1	23		S1	24		S1	24	
Ninth (last)											
M1, M1-3				10	58%						
M2, M2-4, M1-4, M2-5				11	54%						
M2-2, M1-2, M2-3, M1-5				12	50%						
S2				23	4%						
B1, B2				24							
S1				28							

Figure 8.32 continued: Population size=32K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1	11.32	43.23%	M1	11.32	46.10%
M2	11.53	42.18	M2	11.62	44.67
M2-2	12.79	35.86	M1-5	12.43	40.81
M1-4	12.80	35.81	M2-2	12.81	39.00
M1-5	12.81	35.76	M1-4	12.96	38.29
M1-3	13.38	32.90	M2-5	13.06	37.81
M2-5	13.40	32.80	M1-3	13.39	36.24
M2-3	13.71	31.24	M1-2	13.85	34.05
M1-2	13.72	31.19	M2-3	14.11	32.81
M2-4	14.71	26.23	M2-4		
B2	19.72	1.10	B2	18.82	10.38
B1	19.94		S2	18.90	10.00
S1	20.67		S1	19.22	8.48
S2	21.07		B1	21.00	

Figure 8.33: Population size=32K; 30 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.2.7 Summary of experiment results

On the whole, the MTFs tend to outperform the other methods. However, with little consistency in the performance of MTF1- i MTF2- i , $i \in \{2,3,4,5\}$, the focus is being moved to M1 and M2. Based on the experimentation in this section M1 and M2 are almost always more efficient than B1. While M1 appears to perform far better than M2, this observation is not supported by the *U test*. For the two rankings which use the same lists of numbers for the calculations (probability that a method will get a solution first and $G\mu$ for the sample), the degree of confidence that the difference in performance between M2 and B1 when *maxgen* = 100 is extremely low indicating that the difference might be due to chance in this single instance. The SFSs have again shown no advantage over the benchmarks. For the next experiment the population size is again doubled and *maxgen* appropriately reduced.

8.2.2.3 Population size 64K; 30 seeds

For this experiment the population size is the same as the one used by Jefferson et al. (1992). While for the previous two experiments (sections 8.2.2.1 and 8.2.2.2 respectively) data were collected for two values of *maxgen*, for this experiment data were recorded for three different *maxgens*. Timing data were recorded but deemed unusable due to the difference in the processors on which the data were generated. Both M1 and M2 surpass B1 in this experiment. M2 shows an advantage over M1, performing far better when the failure rate is taken into consideration (figures 8.34, 8.35, 8.38, and 8.39).

<i>maxgen</i> = 70				<i>maxgen</i> = 60				<i>maxgen</i> = 50			
M2	3	2.00	3.66	M2	4	2.29	5.65	M2	8	3.48	5.92
M1-2	4	2.29	3.21	M1-2	6	2.86	4.51	M1-3	10	4.19	4.92
M1-4				M1-3				M2-2	11	4.59	4.50
M1-3	6	2.86	2.56	M1-4				M1	12	5.03	4.11
M2-2	7	3.16	2.32	M2-2	8	3.48	3.71	M1-2			
M1	8	3.48	2.10	M1	9	3.82	3.38	M1-4			
M2-4				M2-4				M2-4	13	5.51	3.75
M2-5	9	3.82	1.92	M1-5				M1-5	14	6.04	3.42
M1-5				M2-3	12	5.03	2.57	M2-3	16	7.33	2.82
M2-3	10	4.19	1.75	M2-5	13	5.51	2.34	M2-5			
B2				S1	15	6.64	1.94	S1	22	14.85	1.39
S1	12	5.03	1.46	B2				S2	23	17.33	1.19
S2	14	6.04	1.21	S2	17	8.11	1.59	B1	24	20.64	
B1	16	7.33		B1	21	12.91		B2			

Figure 8.34: Population size=64K; 30 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.3.1 Failure rate and K

Based on the number of failures and K (data in figure 8.34, description in

section 7.2.2.10), the SFS and benchmarks are at the bottom of the lists with no consistency of order. M1 and M2 did well with speedups two to six times that of the benchmark. MTF with competition (M2) far surpasses M1's performance. When the MTFs are applied every i^{th} generation, $i \in \{2,3,4,5\}$, all perform much better than the benchmark (last column of each list of figure 8.34).

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1-3	10.03%	447%	M1-3	16.67%	620%	M1-3	22.77%	744%
M1-2	.003	424	M2	.92	900	M1-2	.90	575
M2	9.64	702	M1-2	6.10	546	M2	13.19	988
M1-4	19.45	472	M1-4	17.69	575	M1-4	6.96	573
M2-2	37.57	348	M2-2	37.79	474	M2-2	41.91	603
M1	4.81	267	M1	2.55	381	M1	5.95	457
M1-5	.48	231	M1-5	5.24	345	M2-3	2.71	347
M2-3	3.38	187	M2-3	2.50	273	M1-5	.07	371
M2-4	37.97	295	M2-4	48.29	419	M2-4	42.88	523
M2-5	48.83	163	M2-5	45.75	210	M2-5	110.14	273
S1	12.38	75	S1	27.54	113	S1	31.52	65
B2	22.43	72	B2	8.83	76	S2	16.33	26
S2	30.72	31	S2	54.58	55	B2	8.50	9
B1			B1			B1		

Figure 8.35: Population size=64K; 30 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is below B1.

8.2.2.3.2 Probability of fewer generations

Based on the probability that one method will get a solution in fewer generations than the next (data in figure 8.35, description in section 7.2.2.10), the SFSs and the benchmarks are the poorest performers appearing near the bottom of these lists. M2 is more likely to get a solution before M1. M1 also does well, but it is nowhere nearly as impressive as M2. B1 is consistently the worst performer.

From the *U Test* almost all the methods across the three *maxgens* have a high degree of confidence that the difference in performance with respect to B1 is significant. The most restrictive results occur when *maxgen* = 50. In this case only the MTFs have a .998 degree of confidence.

8.2.2.3.3 Grouping of best five runs based on number of generations

In figure 8.36, the best five runs based on the number of generations to a solution are grouped and compared. M1 and M2 are consistently above the SFSs and benchmarks with speedups over B1 between 1.27 and 1.67. The SFSs and benchmarks fill the lowest 4 positions of all five lists with S1 consistently positioned above the benchmarks.

Best runs	Second best	Third best	Fourth best	Fifth best
M1-5 22 2.05	M1-3 24 1.92	M1-3 26 1.77	M1-2 28 1.68	M1-3 28 1.68
M1-3 24 1.88	M1-4 26 1.77	M1-2 27 1.70	M1-3	M1-2 29 1.62
M1-4 25 1.80	M1-2 27 1.70	M1-4 30 1.53	M1-4 31 1.52	M2 32 1.47
M1-2 26 1.73	M2 29 1.59	M2 31 1.48	M2 32 1.47	M1-4
M2 27 1.67	M1 30 1.53	M2-3 32 1.44	M2-2 34 1.38	M2-2 34 1.38
M2-3 28 1.61	M1-5	M2-2 33 1.39	M2-3	M2-3 36 1.31
M2-5	M2-2 31 1.48	M2-5 34 1.35	M1-5	M1 37 1.27
M2-2 29 1.55	M2-3	M1-5	M1 37 1.27	M2-4 38 1.24
M1 30 1.50	M2-5 34 1.35	M1 35 1.31	M2-5	M2-5 39 1.21
M2-4 33 1.36	M2-4 36 1.28	M2-4 36 1.28	M2-4 38 1.24	M1-5
S1 37 1.22	S1 38 1.21	S1 40 1.15	S1 43 1.09	S1 44 1.07
B2	B2 42 1.10	S2 45 1.02	S2 46 1.02	S2 46 1.02
S2 39 1.15	S2 45 1.02	B1 46	B2	B1 47
B1 45	B1 46	B2	B1 47	B2

Figure 8.36: Population size=64K; 30 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.3.4 G_{AV}

In figure 8.37, G_{AV} (description in section 7.2.1) is the basis for comparison. The performance of M1 and M2 is close, both having small speedups over B1. M2 is minimally better than M1 on all three lists. M1-3 tops these three lists and, looking back over figures 8.34 through 8.36, it is also near the top of all the previous lists. When $maxgen = 70$, these three MTFs have a .998 degree of confidence that the difference in performance with respect to B1 is significant. For $maxgen = 60$, M1-3 and M2 have a .99 degree of confidence, and M1's degree of confidence is .98. Due to the high failure rate (section 8.8.3.1) for B1 when $maxgen = 50$, there are not enough data points to get dependable *U Test* results with the standard z score table (Freund 1962). The benchmarks and SFSs fill the lowest four positions of all three lists with S1 again consistently positioned before the benchmarks..

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1-3	38.62	1.43	M1-3	38.62	1.29	M1-3	34.90	1.34
M2-2	41.83	1.32	M1-2	40.46	1.23	M1-2	35.61	1.31
M2	42.48	1.30	M2-2	40.95	1.22	M1-4	36.83	1.27
M1-2	42.65	1.30	M1-4	41.04	1.22	M2-3	37.07	1.26
M1-4	42.88	1.29	M2-3	41.06	1.22	M2-2	38.47	1.21
M1-5	42.90	1.29	M2	41.42	1.20	M2	39.18	1.19
M2-3	43.35	1.28	M1-5	42.90	1.16	M1-5	39.31	1.19
M1	44.05	1.26	M1	42.95	1.16	M2-4	40.88	1.14
M2-4	44.50	1.24	M2-4	43.67	1.14	M1	41.06	1.14
M2-5	48.19	1.15	M2-5	44.12	1.13	M2-5	41.43	1.13
S1	50.83	1.09	S1	47.73	1.05	S1	42.75	1.09
S2	53.25	1.04	B1	49.89		B2	44.17	1.06
B2	54.60	1.01	S2	50.23		S2	45.86	1.02
B1	55.36		B2	51.47		B1	46.67	

Figure 8.37: Population size=64K; 30 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.3.5 $G\mu$

In figure 8.38, $G\mu$ is calculated using the formula to predict values for the population of all seeds (section 7.2.2.5), and figure 8.39 contains the data for $G\mu$ using the formula which calculates this value for the sample (section 7.2.2.9). M2 tops the six lists in these two figures with significant speedups over B1. M1 also does well, but its performance is no where nearly as impressive as M2's. The *U Test* for $G\mu$ for the sample (figure 8.39) yields a .998 degree of confidence across the values for *maxgen* that the difference in performance between each of these two MTFs and B1 is significant. The same holds true for the performance of M2 with respect to M1. The benchmarks and SFSs fill the bottom four positions of all six lists, with B1 consistently in the bottom position.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M2	51.37	2.68	M2	51.80	3.73	M2	58.73	4.29
M1-2	54.57	2.52	M1-3	54.87	3.52	M1-3	61.40	4.10
M1-4	54.80	2.51	M1-2	56.71	3.41	M2-2	69.00	3.65
M1-3	57.37	2.40	M1-4	57.29	3.37	M1-2	70.61	3.56
M2-2	64.44	2.13	M2-2	64.13	3.01	M1-4	71.83	3.50
M1	70.87	1.94	M1-5	70.04	2.76	M1	76.06	3.31
M2-4	71.32	1.93	M1	70.09	2.76	M2-4	80.88	3.11
M1-5	74.33	1.85	M2-4	70.81	2.73	M1-5	84.94	2.96
M2-5	79.62	1.73	M2-3	82.73	2.34	M2-3	96.36	2.61
M2-3	79.85	1.72	M2-5	91.77	2.11	M2-5	100.72	2.50
B2	91.10	1.51	S1	109.73	1.76	S1	184.00	1.37
S1	99.16	1.39	B2	113.47	1.70	S2	214.43	1.17
S2	116.38	1.18	S2	131.00	1.48	B2	249.17	1.01
B1	137.50		B1	193.22		B1	251.67	

Figure 8.38: Population size=64K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M2	51.09	2.59	M2	51.46	3.48	M2	57.92	3.84
M1-2	54.17	2.44	M1-3	54.26	3.30	M1-3	60.19	3.70
M1-4	54.40	2.43	M1-2	56.10	3.19	M2-2	67.52	3.30
M1-3	56.66	2.33	M1-4	56.68	3.16	M1-2	68.82	3.23
M2-2	63.54	2.08	M2-2	63.17	2.83	M1-4	70.04	3.18
M1	69.75	1.89	M1-5	68.85	2.60	M1	74.27	3.00
M2-4	70.20	1.88	M1	68.90	2.60	M2-4	78.71	2.83
M1-5	72.95	1.81	M2-4	69.62	2.57	M1-5	82.31	2.70
M2-3	78.16	1.69	M2-3	80.59	2.22	M2-3	92.47	2.41
M2-5	78.24	1.69	M2-5	89.18	2.01	M2-5	96.83	2.30
B2	89.41	1.48	S1	105.92	1.69	S1	168.42	1.32
S1	96.67	1.37	B2	109.66	1.63	S2	193.49	1.15
S2	112.72	1.17	S2	125.30	1.43	B2	220.03	1.01
B1	132.09		B1	178.99		B1	222.53	

Figure 8.39: Population size=64K; 30 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.3.6 Machine size

With respect to machine size, the grouped runs are presented in figure 8.40. With *maxgens* of 70 and 60, even in the **Best runs**, B1 is separated from the rest of the methods. For both *maxgens* the demarcation between the MTFs and the other methods is starting by the **Third best** lists. M1 and M2 remain high on the lists for both *maxgens*, and M2-3 drops away from the other MTFs as the data progress away from the **Best runs**. The other MTFs spread out a bit, but the gap between the MTFs and the other methods remains. When *maxgen*=50, the gap between the MTFs and the other methods appears in the **Second best**, with the chasm widening as the lists move further away from the **Best runs**. When the average machine size is considered (figure 8.41), the MTFs produce the smallest machines. Due to the high failure rate for B1 when *maxgen* = 50 (figure 8.34) there is insufficient data to get a reliable *U Test* using the standard z score table (Freund 1962). For *maxgen* = 70, M1-5 has a .96 degree of confidence, M2-5 has a .99

maxgen = 70

Best runs			Second best			Third best			Fourth best		
M2	8	27%	M2	8	33%	M1-4	8	38%	M2	9	40%
M1			M1			M2	9	31%	M1		
M2-2			M1-3			M1			M2-2		
M1-3			M1-4			M2-2			M1-2		
M1-4			M2-2	9	25%	M1-2			M1-3		
B2			M1-2			M2-3			M1-4		
M1-2	9	18%	M2-3			M1-3			M2-3	10	33%
M2-3			M2-4			M2-4			M2-4		
M2-4			M2-5			M2-5			M2-5		
M2-5			M1-5			M1-5			M1-5		
M1-5			B2			S2	11	15%	B2	13	13%
S2			S2	10	17%	B2	12	8%	S2	14	7%
S1			B1	12		B1	13		S1		
B1	11		S1			S1	14		B1	15	
Fifth best			Sixth best			Seventh best			Eighth best		
M2	9	44%	M2	9	47%	M2	9	50%	M2	9	53%
M2-2			M1-2			M1-3			M1-3		
M1-2			M1-3			M1	10	44%	M1	10	47%
M1-3			M1	10	41%	M2-2			M2-2		
M1-4			M2-2			M1-2			M1-2		
M1	10	38%	M2-4			M2-4			M2-4		
M2-3			M1-4			M1-4			M1-4		
M2-4			M2-5			M1-5			M2-5	11	42%
M2-5			M1-5			M2-5	11	39%	M1-5		
M1-5			M2-3	12	29%	M2-3	14	22%	M2-3	14	26%
B2	13	19%	B2	14	18%	B2	15	17%	B2	16	16%
S2	15	6%	S2	16	6%	B1	18		S1	18	5%
B1	16		B1	17		S2			B1	19	
S1	18		S1	18		S1			S2	20	
Ninth best			Tenth best			Eleventh best			Twelfth best		
M2	10	55%	M2	10	57%	M2	10	57%	M2	11	54%
M2-2			M2-2			M2-2			M2-2		
M1-2			M1-2			M1-2			M1-2		
M1-3			M1-3			M1-3			M1-3		
M2-4			M2-4			M1	11	52%	M1-4		
M1	11	50%	M1	11	52%	M1-4			M1	12	50%
M1-4			M1-4			M2-4	12	48%	M2-4		
M2-5			M2-5			M2-5			M2-5		
M1-5	12	45%	M1-5	12	48%	M1-5			M1-5	13	46%
M2-3	16	27%	M2-3	16	30%	M2-3	16	30%	M2-3	17	29%
B2	19	14%	B2	19	17%	B2	19	17%	S2	21	13%
S2	20	9%	S2	21	9%	S2	21	9%	S1	22	8%
S1			S1			S1			B2		
B1	22		B1	23		B1	23		B1	24	

Figure 8.40: Population size=64K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.) Continued on next page.

maxgen = 70 (continued)

Thirteenth best			Fourteenth (last)		
M2	11	54%	M1-2	11	56%
M1-2			M2	12	52%
M1-3			M2-2		
M1-4			M1-3		
M1	12	50%	M2-4		
M2-2			M1-4		
M2-4			M2-5		
M2-5			M1	13	48%
M1-5	13	46%	M1-5	14	44%
M2-3	17	29%	M2-3	18	28%
S2	22	8%	B2	23	8%
B2			S2	24	4%
B1	24		B1	25	
S1			S1		

maxgen = 60

Best runs			Second best			Third best			Fourth best		
M2	8	27%	M2	8	38%	M1-4	8	47%	M2	9	44%
M1			M1			M2	9	40%	M1		
M2-2			M1-3			M1			M2-2		
M1-3			M1-4			M2-2			M1-2		
M1-4			M2-2	9	31%	M1-2			M1-3		
M1-2	9	18%	M1-2			M2-3			M1-4		
M2-3			M2-3			M1-3			M2-3	10	38%
M2-4			M2-4			M2-4			M2-4		
M2-5			M2-5			M2-5			M2-5		
M1-5			M1-5			M1-5			M1-5		
S2			S2	10	23%	B2	13	13%	B2	13	19%
S1			S1	12	8%	S2	14	7%	S2	15	6%
B2			B2			S1			B1	16	
B1	11		B1	13		B1	15		S1	18	

Fifth best			Sixth best			Seventh best			Eighth best		
M2	9	47%	M2	9	53%	M2	9	61%	M2	9	61%
M2-2			M1-3			M1-3			M1-3		
M1-2			M1	10	47%	M1	10	57%	M1	10	57%
M1-3			M2-2			M2-2			M2-2		
M1-4			M1-2			M1-2			M1-2		
M1	10	41%	M2-4			M2-4			M2-4		
M2-3			M1-4			M1-4			M1-4		
M2-4			M1-5			M1-5			M2-5	11	52%
M2-5			M2-5	11	42%	M2-5	11	52%	M1-5		
M1-5			M2-3	12	37%	M2-3	14	39%	M2-3	14	39%
B2	14	18%	B2	16	16%	B2	19	17%	S2	21	9%
B1	17		S1	18	5%	S2	20	13%	S1		
S2	18		B1	19		S1	21	9%	B2	22	4%
S1			S2	20		B1	23		B1	23	

Figure 8.40 continued: Population size=64K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.) Continued on next page.

maxgen = 60 (continued)

Ninth (last)		
M2, M2-2, M1-2, M1-3, M2-4	10	58%
M1, M1-4	11	54%
M2-5, M1-5	12	50%
M2-3	16	33%
S2	21	13%
S1, B2	22	8%
B1	24	

maxgen = 50

Best runs	Second best	Third best	Fourth best
M2 8 27%	M2 8 38%	M1-4 8 47%	M2 9 47%
M1	M1	M2 9 40%	M1
M2-2	M1-3	M1	M2-2
M1-3	M1-4	M2-2	M1-2
M1-4	M2-2 9 31%	M1-2	M1-3
M1-2 9 18%	M1-2	M1-3	M2-3 10 41%
M2-3	M2-3	M2-4	M2-4
M2-4	M2-4	M2-5	M1-4
M2-5	M2-5	M1-5	M2-5
M1-5	M1-5	M2-3 10 33%	M1-5
S1	S1 12 8%	B1 15	B1 17
B2	B1 13	S2	S1 18
S2 10 9%	S2 14	S1 18	S2 20
B1 11	B2	B2 22	B2 22

Fifth best	Sixth (last)
M2 9 61%	M2 9 63%
M2-2	M1-3
M1-3	M1 10 58%
M1 10 57%	M2-2
M1-2	M1-2
M2-4	M2-4
M1-4	M1-4

Fifth best	Sixth (last)
M2-5	M1-5
M1-5	M2-5 12 50%
S2 21 9%	B1 24
S1	S2
B1 23	B2
B2	S1 26

Figure 8.40 **continued**: Population size=64K; 30 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

degree of confidence, and those methods above M2-5 on the list (figure 8.41) fall within the .998 degree of confidence that the difference in performance with respect

to B1 is significant. When $maxgen = 60$, only M1-4 has a .998 degree of confidence. M1 and M2-5 have a .98 degree of confidence, and those methods on the list between M1-4 and M1 have a .99 degree of confidence. M1 and M2 do well, and while B1 is not at the bottom of the three lists, it lies within the bottom four along with B2 and the SFSs.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1-4	11.96	36.07%	M1-4	11.96	33.15%	M1-3	11.50	33.02%
M2	12.30	34.26	M2	12.35	30.97	M1-4		
M1-2	12.50	33.19	M2-4	12.57	29.74	M2	12.05	29.82
M1-3	12.75	31.85	M2-2	12.73	28.84	M2-4	12.35	28.07
M2-2	12.83	31.43	M1-2	12.75	28.73	M1-2	12.67	26.21
M2-4	12.86	31.27	M1-3			M1	12.78	25.57
M1	12.95	30.79	M1	13.05	27.05	M2-2	13.16	23.35
M2-5	13.38	28.49	M2-5	13.06	27.00	M2-5	13.50	21.37
M1-5	14.86	20.58	M1-5	14.86	16.94	M1-5	13.94	18.81
M2-3	15.80	15.55	M2-3	15.67	12.41	M2-3	15.57	9.32
S2	18.19	2.78	B1	17.89		B1	17.17	
B2	18.70	.05	S2	18.46		S2	18.43	
B1	18.71		B2	19.20		B2	19.00	
S1	19.89		S1	20.40		S1	19.62	

Figure 8.41: Population size=64K; 30 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.3.7 Summary of experiment results

Based on this experiment, M2 is more efficient than M1, which in turn surpasses the benchmarks and SFSs. When the *U Test* is applicable it consistently indicates that the difference in performance of M1 and M2 with respect to B1 is significant with a high degree of confidence. The superiority of M1 and M2 over the benchmarks was also apparent in figure 8.5 (section 8.1.2.3); figure 8.5 contains the data which track the evolutionary progress corresponding to this experiment. Based on the data which reflect the failure rate (figures 8.34, 8.35, 8.38, and 8.39), and using

percent increase and speedup over B1 as measures of performance, it can be concluded that performance increases with decreasing *maxgen* for M1 and M2.

For the remaining three experiments (sections 8.2.2.4 through 8.2.2.6) the set of 30 seeds which was used for the first three experiments is replaced with a set of 100 seeds.

8.2.2.4 Population size 16K; 100 seeds

This experiment implements the parameters of the experiment in section 8.2.2.1, but it uses a set of 100 seeds. These seeds do not replicate any of the 30 seeds used for the previous three experiments. The focus of the evaluation is now on M1 and M2. Both M1 and M2 surpass benchmark and SFS performance. M2 outperforms M1 and performance increases with decreasing *maxgen* when the measure is speedup or percent increase over B1 and when the failure rate is included in the measure of performance.

8.2.2.4.1 Failure rate and K, and probability of fewer generations

Based on the number of failures and K (data in figure 8.42, description in section 7.2.2.10), M2 is the best method. The familiar pattern of MTFs followed by the other methods, and M1 residing between M2 and B1 is repeated. The same pattern appears in figure 8.43 where the methods are ranked based on the probability that a method will get a solution in fewer generations. When applying the *U Test* to the points used to produce the values in figure 8.43 there is a .998 degree of

confidence that the difference in performance between any two of these three methods is significant in the ordering just described.

<i>maxgen</i> = 125				<i>maxgen</i> = 100			
M2	32	4.04	1.85	M2	34	4.27	2.26
M2-2	34	4.27	1.75	M2-4	40	5.03	1.92
M2-5	36	4.51	1.66	M2-2	42	5.31	1.81
M1-5				M1-5	44	5.61	1.72
M2-4	37	4.63	1.61	M1-2	46	5.93	1.62
M1-4	40	5.03	1.49	M1	49	6.46	1.49
M1-2	42	5.31	1.41	M1-4			
M1	43	5.46	1.37	M2-5	50	6.64	1.45
M2-3	46	5.93	1.26	M2-3	54	7.47	1.29
M1-3	47	6.10	1.23	M1-3	56	7.94	1.21
S1	53	7.25	1.03	S1	61	9.32	1.03
B1	54	7.47		B1	62	9.63	
B2	57	8.19		S2	73	14.63	
S2	63	9.97		B2	74	15.29	

Figure 8.42: Population size=16K; 100 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	5.42%	114%	M2	10.79%	133%
M2-2	5.15	105	M2-2	.25	112
M2-4	15.72	98	M2-4	17.19	118
M1-2	2.18	65	M1-2	4.15	78
M1	.03	63	M1	4.03	74
M1-5	4.96	66	M1-5	6.23	68
M2-5	.67	62	M1-4	5.32	58
M1-4	12.51	57	M2-5	5.83	52
M1-3	1.07	40	M1-3	.67	41
M2-3	38.46	41	M2-3	40.06	43
S1	4.41	44	S1	5.36	5
B1	35.23		B1	56.60	
B2	9.09		S2	10.46	
S2			B2		

Figure 8.43: Population size=16K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is below B1.

8.2.2.4.2 Grouping of best five runs based on number of generations

In figure 8.44, the best five runs based on the number of generations to a solution are grouped and compared. M1 and M2 perform similarly with the biggest difference occurring in the **Fifth best** list. (The MTFs are not significantly separated from the other methods in the **Best runs**, and B2 rather than B1 is the last on all of these lists.)

Best runs	Second best	Third best	Fourth best	Fifth best
M2-2 23 1.48	M1-2 30 1.37	M2 32 1.41	M1-2 32 1.47	M2 33 1.45
M2-5 26 1.31	M2-4	M1-2	M2 33 1.42	M1-2
M1-4 27 1.26	M2-2 31 1.32	M1 33 1.36	M1 34 1.38	M2-2 36 1.33
M1-2 28 1.21	M2 32 1.28	M2-2	M2-2 35 1.34	M1 37 1.30
M2-3	M2-5	M1-5 35 1.29	M1-5 36 1.31	M1-5
M2-4 29 1.17	M1 33 1.24	M2-4 37 1.22	M2-4 38 1.24	M2-4 38 1.26
M2 31 1.10	M1-3 34 1.21	M1-4	M1-4	M1-4 40 1.20
M1	M1-5	M2-5 38 1.18	M2-5 39 1.21	M2-5
M1-5 32 1.06	M1-4 36 1.14	M2-3 41 1.10	M2-3 41 1.15	M2-3 42 1.14
B1 34	M2-3 37 1.11	M1-3	M1-3	M1-3
M1-3	S2 39 1.05	B1 45	B1 47	B1 48
S2 36	B1 41	S2	S1 50	S1 51
S1 40	S1 44	S1 48	S2 54	S2 54
B2 48	B2 51	B2 53	B2	B2 58

Figure 8.44: Population size=16K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.4.3 G_{AV}

In figure 8.45, G_{AV} (description in section 7.2.1) is the basis for comparison. The benchmarks and SFSs fill the bottom four positions on the lists and have the same order on both lists. M1 and M2 have similar performance, both having only small speedups over B1. The *U Test* degree of confidence that the difference in performance is significant with respect to B1 is .97 for M2 when $maxgen = 125$ and

for M1 when $maxgen = 100$. For M1 the degree of confidence is .98 when $maxgen = 125$ and only .75 for M2 when $maxgen = 100$.

$maxgen = 125$			$maxgen = 100$		
M2-4	63.54	1.19	M1	59.10	1.15
M1-2	64.64	1.17	M2-2	59.28	1.14
M2	64.99	1.16	M1-3	59.43	1.14
M1	65.05	1.16	M2-4	60.70	1.12
M2-2	65.64	1.15	M1-2	61.31	1.11
M1-3	68.32	1.11	M2-3	61.61	1.10
M2-3	68.63	1.10	M2	63.29	1.07
M1-4	70.88	1.07	M1-4	63.75	1.06
M1-5	72.39	1.05	M2-5	64.38	1.05
M2-5	75.05	1.01	M1-5	66.96	1.01
S1	75.30	1.01	S1	67.03	1.01
B1	75.70		B1	67.82	
S2	82.24		S2	70.85	
B2	91.21		B2	77.12	

Figure 8.45: Population size=16K; 100 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.4.4 G_{μ}

In figure 8.46, G_{μ} is calculated using the formula to predict values for the population of all seeds (section 7.2.2.5), and figure 8.47 contains the data for G_{μ} using the formula which calculates this value for the sample (section 7.2.2.9). M2 tops the four lists in these two figures with speedups over B1 between 1.78 and 2.01. B2 and S2 are at the bottom of these lists, generally performing far worse than B1. On all four lists M1 falls between M2 and B1, and the benchmarks and SFSs fill the bottom four positions. When applying the *U Test* to the points used to produce the values in figure 8.47 there is a .998 degree of confidence that the difference in performance between any two of these three methods is significant.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	125.28	1.79	M2	116.32	2.01
M2-2	131.55	1.71	M2-4	129.03	1.81
M2-4	138.54	1.62	M2-2	133.42	1.75
M1-5	144.26	1.56	M1-5	147.32	1.59
M2-5	146.93	1.53	M1-2	148.35	1.57
M1-4	155.88	1.44	M1	157.14	1.49
M1-2	156.88	1.43	M1-4	161.79	1.44
M1	161.10	1.39	M2-5	166.38	1.40
M2-3	176.96	1.27	M2-3	181.18	1.29
M1-3	181.06	1.24	M1-3	188.98	1.24
S1	218.39	1.03	S1	226.00	1.03
B1	224.61		B1	233.61	
B2	259.23		S2	344.92	
S2	297.78		B2	365.58	

Figure 8.46: Population size=16K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M2	124.42	1.78	M2	115.54	1.99
M2-2	130.58	1.70	M2-4	127.93	1.79
M2-4	137.38	1.61	M2-2	132.18	1.74
M1-5	143.17	1.55	M1-5	145.92	1.57
M2-5	145.83	1.52	M1-2	146.78	1.56
M1-4	154.50	1.43	M1	155.27	1.48
M1-2	155.33	1.43	M1-4	159.92	1.43
M1	159.46	1.39	M2-5	164.40	1.40
M2-3	175.01	1.27	M2-3	178.65	1.28
M1-3	178.99	1.24	M1-3	186.12	1.23
S1	215.43	1.03	S1	222.05	1.03
B1	221.47		B1	229.38	
B2	255.44		S2	335.17	
S2	292.13		B2	354.93	

Figure 8.47: Population size=16K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.4.5 Machine size

With respect to machine size, snapshots from the grouped runs are presented in figure 8.48. For both *maxgens*, runs are selected to show the development of the stratification. In figure 8.49 the average machine size is considered. In both of these figures, by and large, the MTFs separate from the other four methods with M1 and

maxgen = 125

#5 on lists	#9 on lists	#16 on lists	#33 on lists	#37 (last)
All MTFs	M1 9 25%	M2 10 38%	M2 12 52%	M2 12 52%
9 10%	M2-3	M1	M1-3	M1 13 48%
B1 10	M2-4	M1-3	M1-4	M1-3
S2	M1-4	M2-4	M1-5	M1-4
#5 on lists	#9 on lists	#16 on lists	#33 on lists	#37 (last)
S1 12	M1-5	M1-4	M1 13 48%	M1-5
B2 13	M2 10 17%	M1-5	M2-2	M2-2 14 44%
	M2-2	M2-2 11 31%	M2-4	M1-2
	M1-2	M1-2	M1-2 14 44%	M2-3
	M1-3	M2-3	M2-3	M2-5
	M2-5	M2-5	M2-5	M2-4 15 40%
	B1 12	B1 16	S1 24 4%	B1 25
	S2 13	S1 17	B1 25	S1
	S1 14	S2 19	B2	B2 26
	B2 17	B2	S2 26	S2 28

maxgen = 100

#2 on lists	#9 on lists	#16 on lists	#21 on lists	#26 (last)
M2 8 11%	M1 9 25%	M2 10 41%	M2 11 48%	M2 11 52%
M2-2	M2-3	M1	M1	M1-4
M2-4	M1-4	M1-5	M1-3	M1-5
M2-5	M1-5	M2-2 11 35%	M1-4	M1 12 48%
M1-5	M2 10 17%	M1-2	M1-5	M1-3
B1 9	M2-2	M2-3	M2-2 12 43%	M2-4
M1	M1-2	M1-3	M1-2	M2-2 13 43%
M1-2	M1-3	M2-4	M2-3	M1-2
M2-3	M2-4	M1-4	M2-4	M2-3
M1-3	M2-5	M2-5	M2-5	M2-5
M1-4	B1 12	B1 17	B1 21	B1 23
S2	S2 14	S1 19	S1 22	S1 24
S1 11	S1 15	S2 21	S2 25	S2 26
B2 13	B2 18	B2 22	B2	B2 30

Figure 8.48: Population size=16K; 100 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

M2 appearing near the top of all lists. In figure 8.49, all the MTFs, except M2-5 and M2-4 with *maxgen* = 100, fall within the .998 degree of confidence that the difference in performance between each MTF and B1 is significant. The remaining two MTFs with *maxgen* = 100 have a .99 degree of confidence. The benchmarks and SFSs fill the bottom four positions. B2 is most frequently, but not always last, and B1 is never

last.

<i>maxgen</i> = 125			<i>maxgen</i> = 100		
M1-3	12.32	36.53%	M1-3	12.45	32.81%
M1	12.51	35.55	M1	12.73	31.30
M2	13.13	32.35	M2	13.12	29.20
M1-5	13.36	31.17	M1-5	13.36	27.90
M1-4	13.38	31.07	M1-4	13.39	27.74
M2-4	13.92	28.28	M2-3	13.87	25.15
M2-3	13.98	27.98	M2-4	13.90	24.99
M1-2	14.34	26.12	M1-2	14.19	23.42
M2-5	14.53	25.14	M2-5	14.46	21.96
M2-2	14.65	24.52	M2-2	14.78	20.24
S2	18.95	2.37	S2	18.07	2.48
B1	19.41		B1	18.53	
S1	19.72		S1	20.23	
B2	20.49		B2	20.73	

Figure 8.49: Population size=16K; 100 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1.

8.2.2.4.6 Summary of experiment results

The now familiar performance patterns are repeated. M1 and M2 fare better than B1: M2 performs better than M1 when the failure rate is considered, and performance is even better for both methods (M1, M2) when a smaller *maxgen* is used. The *U Test* indicates that there is a high degree of confidence that the difference in performance between each of these MTFs and B1 is significant when the failure rate is considered. The next experiment uses twice the population size of this one and appropriately adjusted values for *maxgen*.

8.2.2.5 Population size 32K; 100 seeds

This experiment implements the parameters utilized by the experiment in section 8.2.2.2 but uses the set of 100 seeds from the previous experiment. In this

experiment M1 and M2 both surpass the benchmark, but M2 doesn't always outperform M1 and there is much less difference in their performance than was evident in the previous experiments.

8.2.2.5.1 Failure rate and K

Based on the number of failures and K (data in figure 8.50, description in section 7.2.2.10), the common pattern of MTFs grouped at the top with the benchmarks and SFSs filling the last four positions again appears. For both *maxgens* M2 performs better than M1.

<i>maxgen</i> = 100				<i>maxgen</i> = 75			
M2-2	23	3.13	2.00	M2-2	33	4.15	2.67
M2	24	3.23	1.94	M1-2	34	4.27	2.60
M1-2	25	3.32	1.89	M2	35	4.39	2.53
M2-5	27	3.52	1.78	M1	37	4.63	2.39
M1	30	3.82	1.64	M2-4	38	4.76	2.33
M2-4				M1-4	39	4.89	2.27
M1-5				M1-5			
M2-3	31	3.93	1.60	M2-3	41	5.17	2.15
M1-3				M2-5	44	5.61	1.98
M1-4	32	4.04	1.55	M1-3	46	5.93	1.87
B2	37	4.63	1.35	S1	59	8.73	1.27
S1	40	5.03	1.25	B2	61	9.32	1.19
B1	48	6.27		S2	64	10.32	1.07
S2	50	6.64		B1	66	11.08	

Figure 8.50: Population size=32K; 100 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.5.2 Probability of fewer generations

Based on the probability that one method will get a solution in fewer generations than the next (data in figure 8.51, description in section 7.2.2.10), B1 is

the poorest performer appearing at the bottom of the lists, with the benchmarks and SFSs again filling the bottom four positions. M1 and M2 are near the top of both lists with M1 coming before M2. When $maxgen=100$ and both are compared to B1 their order is not consistent, but both methods perform well. There is a .998 degree of confidence from the *U test* that the difference in performance between each of these two MTFs and B1 is significant.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1-2	.28%	178%	M1	1.20%	221%
M1	3.32	161	M1-2	4.61	231
M2	2.11	175	M2-4	1.74	207
M2-4	.89	148	M2	1.63	211
M2-2	6.89	178	M2-2	4.19	217
M1-4	3.89	136	M1-4	6.14	187
M1-5	1.46	132	M1-5	1.52	172
M2-3	6.25	122	M2-3	11.44	161
M2-5	2.81	110	M1-3	.62	127
M1-3	46.51	102	M2-5	61.46	124
S1	12.65	38	S1	27.75	38
B2	20.11	22	S2	1.30	8
S2	1.09	1	B2	6.46	6
B1			B1		

Figure 8.51: Population size=32K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is not better than B1.

8.2.2.5.3 Grouping of best five runs based on number of generations

In figure 8.52, the best five runs based on the number of generations to a solution are grouped and compared. The only consistency of note across the five lists is the clustering of the MTFs above the benchmarks and SFSs. M1 and M2 are near the top of the lists. M1's performance is more consistent with speedups over B1 between 1.44 and 1.63. M2 has speedups between 1.28 and 1.83.

Best runs			Second best			Third best			Fourth best			Fifth best		
M1-5	23	1.91	M2	26	1.73	M2	31	1.45	M1	31	1.48	M1	32	1.44
M2	24	1.83	M2-2			M1			M1-5	32	1.44	M2-5	33	1.39
M1-2			M1	28	1.61	M1-2	32	1.41	M2-2	33	1.39	M1-5		
M2-2	25	1.76	M2-3			M2-3			M1-3			M1-3	34	1.35
M1-3			M1-3	29	1.55	M1-5			M2-5			M2-2	35	1.31
M1	27	1.63	M2-4	30	1.50	M2-2	33	1.36	M2	34	1.35	M2-3		
M2-3			M1-4	31	1.45	M1-3			M1-2			M2-4		
M1-4			M2-5			M2-4			M2-3			M2	36	1.28
M2-4	28	1.57	M1-5			M2-5			M2-4			M1-2		
M2-5	29	1.52	M1-2	32	1.41	M1-4	35	1.29	M1-4	35	1.31	M1-4		
S1	35	1.26	S1	39	1.15	S1	39	1.15	S1	40	1.15	S1	40	1.15
S2	41	1.07	B2	43	1.05	B1	45		B1	46		B1	46	
B2			S2	44	1.02	S2			S2	47		B2	47	
B1	44		B1	45		B2	47		B2			S2	48	

Figure 8.52: Population size=32K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M1	51.79	1.33	M1	48.14	1.20
M1-4	53.16	1.29	M2-4	49.11	1.17
M2-4	53.70	1.28	M1-4	49.90	1.16
M1-2	54.89	1.25	M1-2	50.15	1.15
M1-5	55.14	1.24	M2	50.35	1.15
M2	55.20	1.24	M1-3	50.39	1.14
M2-3	55.77	1.23	M2-3	50.75	1.14
M2-2	55.99	1.23	M1-5	51.08	1.13
M1-3	58.88	1.17	M2-2	51.46	1.12
M2-5	60.37	1.14	M2-5	52.00	1.11
S1	64.08	1.07	S1	54.44	1.06
S2	66.00	1.04	S2	57.53	1.00
B1	68.62		B1	57.68	
B2	70.40		B2	61.36	

Figure 8.53: Population size=32K; 100 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.5.4 G_{AV}

In figure 8.53, G_{AV} (description in section 7.2.1) is the basis for comparison.

The performance of M1 and M2 are close, both having small speedups over B1. M1 tops both lists and is only slightly better than M2 on both lists. There is a .998 degree of confidence for M1 with both *maxgens* and for M2 with *maxgen* = 100 that the difference in performance with respect to B1 is significant. For M2 with *maxgen* = 75 the degree of confidence is .99. The last four positions on both lists are S1, S2, B1, and B2 in that order.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M2-2	87.16	1.87	M2-2	89.89	2.29
M2	88.09	1.85	M1-2	90.30	2.28
M1-2	89.56	1.82	M2	92.27	2.23
M1	96.08	1.70	M1	93.77	2.20
M2-4	97.99	1.66	M2-4	96.69	2.13
M2-5	98.73	1.65	M1-4	99.49	2.07
M1-5	99.43	1.64	M1-5	100.67	2.05
M1-4	101.69	1.60	M2-3	104.56	1.97
M2-3	102.15	1.59	M2-5	112.71	1.83
M1-3	105.26	1.55	M1-3	116.13	1.78
B2	130.72	1.25	S1	164.81	1.25
S1	132.41	1.23	B2	181.23	1.14
B1	162.85		S2	193.64	1.06
S2	168.00		B1	206.21	

Figure 8.54: Population size=32K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.5.5 $G\mu$

In figure 8.54, $G\mu$ is calculated using the formula to predict values for the population of all seeds (section 7.2.2.5), and figure 8.55 contains the data for $G\mu$ using the formula which calculates this value for the sample (section 7.2.2.9). M1 and M2 appear in the upper portion of the MTFs with M2 consistently better than M1, but when *maxgen*=75 it is only marginally better than M1. The *U Test* yields a

.998 degree of confidence that the difference in performance between any two of M1, M2, and B1 is significant. The now familiar pattern of benchmarks and SFSs at the bottom again appears with consistent order within each *maxgen*.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M2-2	86.77	1.86	M2-2	89.34	2.26
M2	87.68	1.84	M1-2	89.72	2.25
M1-2	89.11	1.81	M2	91.65	2.20
M1	95.47	1.69	M1	93.08	2.17
M2-4	97.38	1.65	M2-4	95.95	2.11
M2-5	98.22	1.64	M1-4	98.71	2.05
M1-5	98.82	1.63	M1-5	99.89	2.02
M1-4	101.00	1.59	M2-3	103.68	1.95
M2-3	101.50	1.59	M2-5	111.67	1.81
M1-3	104.61	1.54	M1-3	114.95	1.76
B2	129.79	1.24	S1	162.20	1.25
S1	131.31	1.23	B2	178.26	1.13
B1	161.09		S2	189.99	1.06
S2	166.02		B1	201.99	

Figure 8.55: Population size=32K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.5.6 Machine size

With respect to machine size, snapshots from the grouped runs are presented in figure 8.56. For both *maxgens* runs are selected to show the development of the stratification. The better runs indicate only small differences between the methods. Stratification occurs as the groupings move away from the best runs. As expected the MTFs are at the top with a gap developing between the MTFs and the other methods. On any one list the FSMs found by M1 and M2 differ in size by at most one state. The pattern is the same when the average machine size is considered (figure 8.57), with the differences generally less pronounced. The *U Test* yields a .998 degree of confidence that the difference in performance between each of the MTFs

(not just M1 and M2) and B1 is significant.

maxgen = 100

#6 on lists	#13 on lists	#20 on lists	#35 on lists	#50 (last)
All others	M1-3 9 40%	M1 10 41%	M1-2 11 50%	M2-3 13 52%
9 18%	M1-4	M2-2	M1-3	M1-3
B1 11	M2 10 33%	M1-2	M2-4	M2-4
B2	M1	M2-3	M1-4	M2 14 48%
	M2-2	M1-3	M2-5	M2-2
	M1-2	M2-4	M2 12 45%	M1-2
	M2-3	M1-4	M1	M2-5
	M2-4	M2-5	M2-2	M1 15 44%
	M2-5	M1-5	M2-3	M1-4
	M1-5	M2 11 35%	M1-5	M1-5 16 41%
	S1 11 27%	S1 13 24%	S1 20 9%	B2 23 15%
	S2 14 7%	B1 17	B2 21 5%	S1 24 11%
	B1 15	B2	B1 22	B1 27
	B2	S2	S2 23	S2 30

maxgen = 75

#6 on lists	#9 on lists	#15 on lists	#30 on lists	#34 (last)
All MTFs	M1 9 40%	M1 10 44%	M1-2 11 58%	M1-3 11 62%
9 36%	M2-2	M2-2	M2-3	M1-4
S2 10 29%	M1-2	M1-2	M1-3	M1 12 59%
S1	M2-3	M2-3	M2-4	M1-2
B2 12 14%	M1-3	M1-3	M1-4	M2-3
B1 14	M2-4	M2-4	M2-5	M2-4
	M1-4	M1-4	M1-5	M2-5
	M2-5	M2-5	M2 12 54%	M1-5
	M2 10 33%	M1-5	M1	M2 13 55%
	M1-5	M2 11 39%	M2-2	M2-2
	S1	S1 14 22%	S1 22 15%	S1 24 17%
	S2 14 7%	S2 17 6%	B2	B2
	B2	B1 18	B1 26	S2 27 7%
	B1 15	B2	S2	B1 29

Figure 8.56: Population size=32K; 100 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

maxgen = 100

M1-3	12.04	35.92%
M2-3	12.87	31.51
M2	13.04	30.60

maxgen = 75

M1-3	11.72	39.09%
M2-3	12.42	35.45
M1-4	13.02	32.33

Figure 8.57: Population size=32K; 100 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1. Continued on next page.

<i>maxgen</i> = 100			<i>maxgen</i> = 75		
M2-4	13.09	30.34	M1-2	13.06	32.12
M2-5	13.15	30.02	M2	13.09	31.96
M1-4	13.16	29.96	M2-5	13.12	31.81
M1-2	13.25	29.48	M2-4	13.18	31.50
M1	13.51	28.10	M1	13.46	30.04
M1-5	13.91	25.97	M1-5	13.72	28.69
M2-2	14.03	25.33	M2-2	13.91	27.70
S1	17.58	6.44	S1	17.66	8.21
S2	18.68	.59	S2	18.58	3.43
B1	18.79		B2	18.92	1.66
B2	19.25		B1	19.24	

Figure 8.57 **continued**: Population size=32K; 100 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1.

8.2.2.5.7 Summary of experiment results

Based on this experiment M1 and M2 are preferable, but generally their performance does not differ drastically when speedup or percent increase over B1 is the measure and the failure rate is included in the measure. The *U Test* yields a high degree of confidence that the difference in performance between each of these two MTFs and B1 is significant.

The next experiment is the last in this series. It implements a population size twice the size of the one in this experiment and examines the results for three different values of *maxgen*.

8.2.2.6 Population size 64K; 100 seeds

This experiment implements the same parameters utilized in the experiment in section 8.2.2.3, but uses the 100 seeds from the previous experiment. The population size is the same as that used by Jefferson et al. (1992). Data are recorded for three different *maxgens*. The familiar patterns of M2 better than M1 which in

turn is better than B1 and increased efficiency with decreased *maxgen* are repeated when failure rate is part of the measure.

8.2.2.6.1 Failure rate and K

Based on the number of failures and K (data in figure 8.58. description in section 7.2.2.10), the SFS methods perform better than the benchmarks, but all four perform worse than the MTFs. MTF with competition (M2) surpasses M1's performance. The maximum speedups for both occurs when *maxgen*=50 and attains values which are considered significant.

<i>maxgen</i> = 70				<i>maxgen</i> = 60				<i>maxgen</i> = 50			
M1-2	16	2.51	2.23	M1-2	19	2.77	3.25	M2	33	4.15	6.82
M1-3	17	2.60	2.16	M2	26	3.42	2.64	M1-2	38	4.76	5.95
M2	18	2.69	2.09	M1-3				M1	39	4.89	5.79
M2-3	22	3.04	1.84	M2-5	29	3.72	2.42	M1-3	42	5.31	5.34
M2-5				M1-5				M2-3	46	5.93	4.78
M1-4	28	3.62	1.55	M2-3	31	3.93	2.29	M2-4			
M1	29	3.72	1.51	M1	36	4.51	2.00	M1-5			
M2-4	31	3.93	1.43	M2-2				M2-2	47	6.10	4.65
M2-2	33	4.15	1.35	M2-4				M1-4	49	6.46	4.39
S2	40	5.03	1.12	M1-4	38	4.76	1.89	M2-5			
S1				S2	52	7.04	1.28	S2	67	11.50	2.46
B1	44	5.61		S1	53	7.25	1.24	S1	72	14.02	2.02
B2	45	5.77		B2	57	8.19	1.10	B1	85	28.34	
				B1	60	9.02		B2	88	36.02	

Figure 8.58: Population size=64K; 100 seeds. Ranked by number of failures. The first number following the method name is the number of failures. The second is K; that is there is a 99% chance of getting a solution in K runs. The third number is the speedup over the benchmark (B1) based on K; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.6.2 Probability of fewer generations

Based on the probability that one method will get a solution in fewer generations than the next (data in figure 8.59, description in section 7.2.2.10), M2

is most likely to get a solution first. M1 is also near the top of these lists, but nowhere as impressive as M2. The maximum benefit occurs with the smallest value of *maxgen* for which data were collected. Based on the *U Test* the degree of confidence is .998 that the difference in performance between any two of M1, M2, and B1 is significant. The benchmarks and SFSs are again in the bottom four positions with S2 above S1 and both above the benchmarks.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M2	10.36%	283%	M2	8.86%	352%	M2	21.30%	721%
M1-2	14.98	333	M1-2	17.60	410	M1-2	.84	585
M1-3	4.79	224	M1-3	2.90	271	M1	8.77	580
M1	1.37	179	M1	1.90	242	M1-3	7.52	510
M1-5	9.28	204	M1-5	8.82	259	M1-5	1.55	456
M2-5	2.06	197	M2-2	1.32	200	M2-2	11.47	440
M2-2	4.00	142	M2-5	7.27	244	M1-4	3.03	401
M1-4	.60	145	M1-4	.51	188	M2-5	1.80	390
M2-3	8.90	184	M2-3	5.74	232	M2-4	1.07	400
M2-4	75.32	134	M2-4	92.32	185	M2-3	115.31	397
S2	3.43	34	S2	6.05	49	S2	18.07	139
S1	28.89	29	S1	40.38	40	S1	100.01	100
B1	3.21		B2	3.07	3	B1	33.33	
B2			B1			B2		

Figure 8.59: Population size=64K; 100 seeds. Ranked using the probability that a method will get a solution first; based on the number of generations to a solution including the number of generations processed for failed runs. The first number following the method name is the percent increase over the next method on the list. The second is the percent increase over the benchmark; this is omitted when a method is not better than B1.

8.2.2.6.3 Grouping of best five runs based on number of generations

In figure 8.60, the best five runs based on the number of generations to a solution are grouped and compared. M2 is consistently at the top of the lists with speedups over B1 between 1.48 and 1.76. Except for the **Best runs**, the benchmarks and SFSs fill the bottom four positions of the lists.

Best runs			Second best			Third best			Fourth best			Fifth best		
M2	21	1.76	M2	24	1.54	M2	24	1.54	M2	25	1.48	M2	27	1.48
M1-5	23	1.61	M1	27	1.37	M1	28	1.32	M1-2	28	1.32	M1-2	29	1.38
M1	24	1.54	M1-2			M1-2			M2-3	29	1.28	M2-3		
M2-5			M1-3			M1-3			M1-3			M1-3		
M1-3	25	1.48	M2-4			M2-3	29	1.28	M1	30	1.23	M1	30	1.33
M1-2	26	1.42	M2-5			M2-5			M2-2	31	1.19	M2-2	31	1.29
S2			M1-5			M2-2	30	1.23	M2-4			M2-4	32	1.25
M2-4	27	1.37	M2-2	29	1.28	M2-4			M1-5			M1-4		
M2-3	28	1.32	M2-3			M1-4	31	1.19	M1-4	32	1.16	M2-5		
M2-2	29	1.28	M1-4	30	1.23	M1-5			M2-5			M1-5		
M1-4	30	1.23	S2	36	1.03	B1	37		B1	37		S1	38	1.05
S1	36	1.03	B1	37		S2			S1			S2	39	1.03
B1	37		S1			S1			S2			B1	40	
B2	38		B2	41		B2	42		B2	43		B2	44	

Figure 8.60: Population size=64K; 100 seeds. Ranked by the generation number in which a solution was found—grouped runs. The first number following the method name is the generation number. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1	42.13	1.29	M1	39.77	1.28	M2	38.18	1.13
M2	42.40	1.29	M2	39.85	1.27	M2-2	38.79	1.11
M2-2	42.51	1.28	M2-2	41.61	1.22	M1-5	38.87	1.11
M1-2	43.38	1.26	M1-4	42.00	1.21	M1	39.10	1.10
M1-5	44.68	1.22	M1-2	42.54	1.19	M1-2	39.19	1.10
M2-4	44.88	1.21	M1-5	42.58	1.19	M1-3		
M1-4	45.21	1.21	M1-3	42.92	1.18	M1-4	39.51	1.09
M1-3	45.53	1.20	M2-4	43.34	1.17	M2-5	39.80	1.08
M2-5	45.62	1.20	M2-5	43.77	1.16	M2-4	41.15	1.04
M2-3	46.53	1.17	M2-3	43.80	1.16	M2-3	41.17	1.04
S2	50.87	1.07	S2	47.31	1.07	B1	43.00	
S1	51.57	1.06	S1	48.09	1.06	S1	43.29	
B1	54.52		B1	50.80		S2	43.61	
B2	55.22		B2	52.65		B2	45.33	

Figure 8.61: Population size=64K; 100 seeds. Ranked by G_{AV} . The first number following the method name is G_{AV} . The second is the speedup over the benchmark; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

8.2.2.6.4 G_{AV}

In figure 8.61, G_{AV} (description in section 7.2.1) is the basis for comparison.

The performances of M1 and M2 are very close, both having small speedups over B1.

The speedups decrease with *maxgen*. For *maxgen* = 60 and 70 the degree of confidence based on the *U Test* that the difference in performance between of these two MTFs and B1 is significant is .998. When *maxgen* = 50, for M2 the degree of confidence is .98 and for M1 it is .97. The benchmarks and SFSs fill the bottom four positions of all three lists.

8.2.2.6.5 $G\mu$

In figure 8.62, $G\mu$ is calculated using the formula to predict values for the population of all seeds (section 7.2.2.5), and figure 8.63 contains the data for $G\mu$ using the formula which calculates this value for the sample (section 7.2.2.9). For the latter, based on the *U Test*, the degree of confidence that the difference in performance between any two of M1, M2, and B1 is significant. M2 outperforms M1 and all MTFs outperform the other four methods.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1-2	57.90	1.92	M1-2	57.85	2.48	M2	64.30	5.18
M2	58.99	1.89	M2	62.28	2.30	M1-2	71.45	4.66
M1-3	61.07	1.82	M1-3	65.35	2.19	M1	72.71	4.58
M1-5	65.71	1.69	M1-5	68.50	2.09	M1-3	77.12	4.32
M2-5	66.65	1.67	M2-5	69.69	2.06	M1-5	83.31	4.00
M2-3	67.56	1.65	M2-3	72.21	1.98	M2-2	85.02	3.92
M1	72.13	1.54	M1	75.08	1.91	M2-4	85.59	3.89
M1-4	73.82	1.51	M2-2	76.92	1.86	M2-3	85.61	3.89
M2-4	77.78	1.43	M2-4	78.65	1.82	M1-4	89.51	3.72
M2-2	78.48	1.42	M1-4	80.39	1.78	M2-5	89.80	3.71
S2	99.20	1.12	S2	114.39	1.25	S2	148.16	2.25
S1	99.90	1.11	S1	117.88	1.22	S1	175.43	1.90
B1	111.31		B2	134.51	1.07	B1	333.00	
B2	114.31		B1	143.30		B2	420.33	

Figure 8.62: Population size=64K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the population. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M1-2	57.74	1.91	M1-2	57.67	2.45	M2	63.93	4.93
M2	58.80	1.88	M2	62.00	2.28	M1-2	70.95	4.44
M1-3	60.90	1.81	M1-3	65.07	2.17	M1	72.18	4.36
M1-5	65.45	1.69	M1-5	68.15	2.07	M1-3	76.50	4.12
M2-5	66.39	1.66	M2-5	69.34	2.03	M1-5	82.52	3.82
M2-3	67.30	1.64	M2-3	71.81	1.96	M2-2	84.18	3.74
M1	71.73	1.54	M1	74.55	1.89	M2-4	84.80	3.71
M1-4	73.44	1.50	M2-2	76.39	1.85	M2-3	84.82	3.71
M2-4	77.32	1.43	M2-4	78.12	1.81	M1-4	88.57	3.56
M2-2	77.97	1.42	M1-4	79.79	1.77	M2-5	88.86	3.54
S2	98.43	1.12	S2	113.04	1.25	S2	145.11	2.17
S1	99.13	1.11	S1	116.44	1.21	S1	170.91	1.84
B1	110.33		B2	132.67	1.06	B1	314.94	
B2	113.27		B1	141.07		B2	391.56	

Figure 8.63: Population size=64K; 100 seeds. Ranked by $G\mu$ calculated using the formula for the sample. The first number following the method name is $G\mu$. The second is the speedup over the benchmark; this is omitted when a method is not faster than B1.

8.2.2.6.6 Machine size

With respect to machine size, snapshots from the grouped runs are presented in figure 8.64. For each *maxgen* runs are selected to show the development of the stratification. When *maxgen*=70 the gap between the MTFs and the other four methods appears in #21 on lists. When *maxgen*=60 the pattern is repeated with the gap appearing in #19 on lists. For both *maxgens* the MTFs spread out a bit as the runs move towards the last lists. When *maxgen*=50 there are only 12 lists and on the last of these the stratification is just beginning with the gap appearing between the SFSs and the benchmarks. When the average machine size is considered (figure 8.65), the MTFs produce the smallest machines and the other four methods the largest. M1 and M2 are very close in performance and appear close to the top of all three lists. Improvement of these two methods occurs with decreasing *maxgen*. The *U Test* yields a .998 degree of confidence that the difference in performance between each of the MTFs (not just M1 and M2) and B1 is significant.

maxgen = 70

#12 on lists	#16 on lists	#21 on lists	#36 on lists	#55 (last)
All MTFs	M2 9 36%	All MTFs	M2 11 48%	M2 13 52%
S2 9 31%	M2-2	10 38%	M1	M1-2 14 48%
S1 10 23%	M1-3	S2 12 25%	M2-2	M1-3
B1 11 15%	M1-4	B1 16	M1-2	M1-4
B2 13	M1 10 29%	S1	M1-3	M2-5
	M1-2	B2 17	M1-4	M1-5
	M2-3		M2-5	M1 15 44%
	M2-4		M1-5	M2-4
	M2-5		M2-3 12 43%	M2-2 16 41%
	M1-5		M2-4	M2-3
	S2 11 21%		S2 18 14%	S2 26 4%
	S1 13 7%		B1 21	S1
	B1 14		B2	B1 27
	B2 15		S1 22	B2 30

maxgen = 60

#9 on lists	#14 on lists	#19 on lists	#31 on lists	#40 (last)
M2-2 8 47%	M2 9 47%	All MTFs	M2 10 57%	M2 11 63%
M2 9 40%	M2-2	10 33%	M1 11 52%	M2-2
M1	M1-2	S2 12 37%	M2-2	M2-5
M1-2	M1-3	B2 18 5%	M1-2	M1 12 60%
M2-3	M1-4	B1 19	M2-3	M1-2
M1-3	M1 10 41%	S1	M1-3	M1-3
M2-4	M2-3		M2-4	M1-4
M1-4	M2-4		M1-4	M1-5
M2-5	M2-5		M2-5	M2-3 13 57%
M1-5	M1-5		M1-5	M2-4
S2	S2 11 35%		S2 18 22%	S2 24 20%
S1 11 27%	S1 13 24%		B2 22 4%	S1 25 17%
B2 13 13%	B2 16 6%		B1 23	B2 27 10%
B1 15	B1 17		S1	B1 30

maxgen = 50

#12 (last)	
M2, M1, M2-2, M1-2, M1-3, M1-4	9 63%
M2-3, M2-4, M2-5, M1-5	10 58%
S2	11 54%
S1	12 50%
B1	24
B2	28

Figure 8.64: Population size=64K; 100 seeds. Ranked by machine size—grouped runs. The first number following the method name is the machine size. The second is the percent decrease from B1; this is omitted when a method is not faster than B1. (When no data follow the method name, the data are identical to the line immediately above and the two methods are ranked identically.)

<i>maxgen</i> = 70			<i>maxgen</i> = 60			<i>maxgen</i> = 50		
M2-2	12.33	32.51%	M2-2	12.12	36.14%	M2-2	11.72	42.18%
M1-4	12.62	30.93	M2	12.58	33.72	M1	12.56	38.04
M1	12.70	30.49	M1	12.61	33.56	M2	12.66	37.54
M2	12.74	30.27	M1-4	12.68	33.19	M1-4	12.69	37.40
M2-4	12.94	29.17	M2-4	12.72	32.98	M1-5	12.78	36.95
M1-5	13.09	28.35	M1-3	13.00	31.51	M2-4	12.80	36.85
M1-3	13.22	27.64	M2-5	13.14	30.77	M1-3	13.03	35.72
M2-5	13.44	26.44	M1-5	13.23	30.30	M2-5	13.14	35.18
M1-2	13.61	25.51	M2-3	13.58	28.45	M2-3	13.50	33.40
M2-3	13.64	25.34	M1-2	13.59	28.40	M1-2	13.71	32.36
S2	16.37	10.40	S2	15.77	16.91	S2	15.79	22.10
B1	18.27		B2	18.58	2.11	B2	16.25	19.83
B2	18.42		S1	18.77	1.11	S1	17.68	12.78
S1	18.95		B1	18.98		B1	20.27	

Figure 8.65: Population size=64K; 100 seeds. Ranked by average machine size. The first number following the method name is the average machine size. The second is the percent decrease from B1.

8.2.2.6.7 Summary of experiment results

The performance pattern for this experiment appears also in most of the other experiments; M2 outperforms M1 which outperforms B1. The *U Test* indicates that there is a high degree of confidence (usually $\geq .95$) that the difference in the performance of M1 and M2 with respect to B1 is significant. The SFSs have consistently fallen within the bottom four positions of the lists along with B1 and B2. The next section presents the data presented in sections 8.2.2.1 through 8.2.2.6 from a different perspective: the data for a given criterion are viewed across the six experiments.

8.2.3 Evaluation of individual criterion

In viewing the data within each experiment, it becomes clear that certain criteria are better measures than others of all over efficiency, and some measures are equivalent. For example, the ranking based on the timing data generally reflects the

ranking based on the corresponding data for the number of generations (except perhaps when the performance of two methods is very close). While the two criteria induce similar rankings, the speedups or percent increased are somewhat smaller for the timing data reflecting the additional work added to B1 to implement the other methods.

G_{AV} (description in section 7.2.1) adds at most $maxgen + 1$ generations to the number of generations expected due to the number of failures preceding the successful run. When several failed runs precede a successful run, the failure rate is the predominant factor in determining the number of generations which will be needed for the GA to locate a solution. Thus, those measures which take the failure rate into account are considered better indicators of all over performance. The grouped runs (of section 8.2.2) compare the best runs from each method, but this criterion does not take into account the different failure rates.

The data presented with respect to machine size are consistent across the six experiments. As hypothesized, on the average the MTFs produce smaller machines than the SFSs and benchmarks. With respect to the grouped runs, the better runs show little difference between the methods, but as the groupings move away from the **Best runs** stratification occurs with the MTFs producing smaller machines than the other methods. While this is not considered a measure of efficiency per se it does enhance the preference for M1 and M2.

Taking all this into account and the fact that the timing data are available for only one experiment, the following subsections will reexamine G_{AV} (section 8..2.3.1),

the failure rate and K (section 8.2.3.2), the probability of obtaining a solution in fewer generations (section 8.2.3.3), and the two $G_{\mu s}$ (section 8.3.3.4). In these sections the data are reduced to the six main methods (B1, B2, S1, S2, M1, and M2), and speedup or percent increase over B1. When the *U Test* is applicable and the degree of confidence that a method is better than B1 is $\geq .95$, the degree of confidence is also listed.

8.2.3.1 G_{AV}

G_{AV} is the average generation in which a solution is found. Hence G_{AV} is a measure only for successful runs; the failure rate and its effect on the average number of generations that a GA can be expected to examine is ignored. Speedups tend to be smaller than when the failure rate is taken into account. This is because there is a much smaller range of values which will be averaged to get a value for G_{AV} . Another point to consider is that G_{AV} decreases with *maxgen*. This follows from the fact that reducing the value of *maxgen* eliminates the highest values from the average. For example, by lowering *maxgen* from 100 to 75, those runs which found a solution after generation 75 are removed from the list of numbers which are averaged for G_{AV} when *maxgen* is 100.

Figure 8.66 contains a summary of the ranking and speedups induced by G_{AV} across the six experiments. M1 and M2 are at the top of all the lists and generally perform far better than the other four methods. Except for the 32K, 30-seed experiment, M2 has higher speedups than M1, but sometimes the difference is

negligible. The *U Test* generally supports the superiority of M1 and M2 over B1 with a high degree of confidence.

16K, 30 seed <i>maxgen</i> = 125	16K, 30 seed <i>maxgen</i> = 100	16K, 100 seed <i>maxgen</i> = 125	16K, 100 seed <i>maxgen</i> = 100
M1 1.35 .99	M1 1.34 .99	M2 1.16 .98	M1 1.15 .98
M2 1.33 .99	M2 1.29 .99	M1 1.16 .99	M2 1.07
S1 1.01 *	S1 1.13 *	S1 1.01	S1 1.01
B2 1.01	B2 1.05	B1	B1
B1	S2 1.02 *	S2	S2
S2	B1	B2	B2
32K, 30 seed <i>maxgen</i> = 100	32K, 30 seed <i>maxgen</i> = 75	32K, 100 seed <i>maxgen</i> = 100	32K, 100 seed <i>maxgen</i> = 75
M1 1.53 .999	M1 1.37 .999	M1 1.33 .999	M1 1.20 .999
M2 1.33 .99	M2 1.26 .99	M2 1.24 .999	M2 1.15 .99
S1 1.13	S2 1.21 .96	S1 1.07	S1 1.06
S2 1.11	S1 1.16	S2 1.04	S2 1.00
B1	B2 1.07	B1	B1
B2	B1	B2	B2
64K, 30 seed <i>maxgen</i> = 70	64K, 30 seed <i>maxgen</i> = 60	64k, 30 seed <i>maxgen</i> = 50	
M2 1.30 .999	M2 1.20 .99	M2 1.19 *	
M1 1.26 .999	M1 1.16 .99	M1 1.14 *	
S1 1.09	S1 1.05	S1 1.09 *	
S2 1.04	B1	B2 1.06 *	
B2 1.01	B2	S2 1.02 *	
B1		B1	
64K, 100 seed <i>maxgen</i> = 70	64K, 100 seed <i>maxgen</i> = 60	64K, 100 seed <i>maxgen</i> = 50	
M1 1.29 .999	M1 1.28 .999	M2 1.13 .99	
M2 1.29 .999	M2 1.27 .999	M1 1.10 .98	
S2 1.07 .98	S2 1.07 .98	B1	
S1 1.06 .95	S1 1.06	S1	
B1	B1	S2	
B2	B2	B2	

Figure 8.66: Summary for ranking by G_{AV} , with speedup over B1. Data condensed from figures 8.18, 8.29, 8.37, 8.45, 8.53, and 8.61. The second number on each list is the degree of confidence that the performance of the given method is better than B1 when the degree of confidence is $\geq .95$. * indicates an insufficient number of data points for a reliable *U Test*.

While G_{AV} looks only at the successful runs, K (description in section 7.2.2.10) only considers the failure rate. The next section reviews the data based on

the failure rate and K.

16K. 30 seed <i>maxgen</i> = 125	16K. 30 seed <i>maxgen</i> = 100	16K. 100 seed <i>maxgen</i> = 125	16K. 100 seed <i>maxgen</i> = 100
M2 2.83	M2 3.26	M2 1.85	M2 2.26
M1 1.93	M1 2.26	M1 1.37	M1 1.49
B2 1.22	B2 1.13	S1 1.03	S1 1.03
B1	B1	B1	B1
S2	S2	B2	S2
S1	S1	S2	B2
32K. 30 seed <i>maxgen</i> = 100	32K. 30 seed <i>maxgen</i> = 75	32K. 100 seed <i>maxgen</i> = 100	32K. 100 seed <i>maxgen</i> = 75
M1 1.58	M1 2.33	M2 1.94	M2 2.53
B2 1.10	M2 1.34	M1 1.64	M1 2.39
B1	B1	B2 1.35	S1 1.27
M2	B2	S1 1.25	B2 1.19
S2	S2	B1	S2 1.07
S1	S1	S2	B1
64K. 30 seed <i>maxgen</i> = 70	64K. 30 seed <i>maxgen</i> = 60	64K. 30 seed <i>maxgen</i> = 50	
M2 3.66	M2 5.65	M2 5.92	
M1 2.10	M1 3.38	M1 4.11	
B2 1.75	S1 1.94	S1 1.39	
S1 1.46	B2 1.94	S2 1.19	
S2 1.21	S2 1.59	B1	
B1	B1	B2	
64K. 100 seed <i>maxgen</i> = 70	64K. 100 seed <i>maxgen</i> = 60	64K. 100 seed <i>maxgen</i> = 50	
M2 2.09	M2 2.64	M2 6.82	
M1 1.51	M1 2.00	M1 5.79	
S2 1.12	S2 1.28	S2 2.46	
S1 1.12	S1 1.24	S1 2.02	
B1	B2 1.10	B1	
B2	B1	B2	

Figure 8.67: Summary for ranking by number of failures and K, with speedup over B1 based on K
Data condensed from figures 8.13, 8.26, 8.34, 8.42, 8.50, and 8.58.

8.2.3.2 Failure rate and K

In the previous section, without consideration of the failure rate, M1 and M2 were most efficient based on the successful runs. However, the failure rate has more

of an impact on the average number of generations that a GA will need to find a solution. In figure 8.67 speedup over B1 is computed using K (description in section 7.2.2.10). The speedups are based on a 99% chance that a GA will find a solution in K runs. Once the degree of certainty is fixed, K is a function solely of the failure rate.

With a completely different criterion from that of the previous section, the superiority of M1 and M2 is again apparent as indicated in figure 8.67. Except for the 32K, 30-seed experiment, M2 generally has much higher speedups than M1. Except for *maxgen*=100 for the 32K, 30 seed experiment, M1 and M2 top the lists.

Based on the first two very different criteria, M1 and M2 usually get solutions faster than the other methods, with M2 generally performing better than M1. In the next two sections (8.2.2.3 and 8.2.2.4), the number or average number of generations for successful runs is combined with the failure rate for ranking. With M1 and M2 ranked best based on these two separate criteria, it is expected that they will be similarly ranked when the two criteria are combined. The next two sections (8.2.2.3 and 8.2.2.4) analyze the data when the failure rate and the data for the successful runs are combined.

8.2.2.3 Probability of fewer generations

The previous two sections separately considered two different criteria: the average number of generations expected for a successful run, and the failure rate.

16K, 30 seed <i>maxgen</i> = 125	16K, 30 seed <i>maxgen</i> = 100	16K, 100 seed <i>maxgen</i> = 125	16K, 100 seed <i>maxgen</i> = 100
M2 353% .999	M2 398% .999	M2 114% .999	M2 133% .999
M1 192 .999	M1 229 .999	M1 63 .999	M1 74 .999
B2 20 .99	B2 16	S1 44	S1 5
B1	B1	B1	B1
S2	S2	B2	S2
S1	S1	S2	B2
32K, 30 seed <i>maxgen</i> = 100	32K, 30 seed <i>maxgen</i> = 75	32K, 100 seed <i>maxgen</i> = 100	32K, 100 seed <i>maxgen</i> = 75
M1 345% .999	M1 472% .999	M2 175% .999	M1 221% .999
M2 62	M2 95 .999	M1 161 .999	M2 211 .999
B2 6	B1	S1 38 .999	S1 38 .999
B1	B2	B2 22 .999	S2 8 .96
S2	S2	S2 1	B2 6 .999
S1	S1	B1	B1
64K, 30 seed <i>maxgen</i> = 70	64K, 30 seed <i>maxgen</i> = 60	64K, 30 seed <i>maxgen</i> = 50	
M2 702% .999	M2 900% .999	M2 988% .999	
M1 267 .999	M1 381 .999	M1 457 .999	
S1 75 .999	S1 113 .999	S1 65	
B2 72 .999	B2 76 .999	S2 26	
S2 31 .99	S2 55 .999	B2 9	
B1	B1	B1	
64K, 100 seed <i>maxgen</i> = 70	64K, 100 seed <i>maxgen</i> = 60	64K, 100 seed <i>maxgen</i> = 50	
M2 283% .999	M2 352% .999	M2 721% .999	
M1 179 .999	M1 242 .999	M1 580 .999	
S2 34 .999	S2 49 .999	S2 139 .999	
S1 29 .999	S1 40 .999	S1 100 .999	
B1	B2 3 .99	B1	
B2	B1	B2	

Figure 8.68: Summary for ranking for the probability that a method will get a solution first, with percent increase over B1. Data condensed from figures 8.14, 8.27, 8.35, 8.43, 8.51, and 8.59. The second number on each list is the degree of confidence that the performance of the given method is better than B1 when the degree of confidence is $\geq .95$.

The probability that a method will get a solution in fewer generations combines these two criteria (description in section 7.2.2.10). For this criterion the ranking is based on the percent increase over the benchmark. Since M1 and M2 generally surpassed the other four methods, it is not a surprise that they top all the lists in figure 8.68. M2's performance is far better than M1's except for the 32K experiments; but this

reversal is also evident in figures 8.66 and 8.67 of the previous subsection. In all but one case (32K, 30 seed, $maxgen = 100$, M2), the *U Test* supports the superiority of M1 and M2 over B1 with a high degree of confidence.

The rankings based on the three criteria examined so far generally favor M2 over M1, and M1 over B1. In the next section two more sets of data are examined. Each of them combines the failure rate with the average number of generations for a successful run.

8.2.3.4 The $G\mu$ s

While the validity of the two $G\mu$ s (sections 7.2.5, 7.2.8, and 7.2.9) as a measure of efficiency was questionable due to the high standard deviations associated with them, the *U Test* can be used to obtain a degree of confidence for the conclusions for the $G\mu$ for the sample. Also by comparing the rankings induced by the $G\mu$ s with those induced by the criteria in sections 8.2.3.1 through 8.2.3.3 the usefulness of both of the $G\mu$ s as criteria for ranking can be determined.

$G\mu$ is the mean number of generations which should be needed to find a solution based on the data from an experiment consisting of several runs. It is the sum of the average number of generations for successful runs and the mean number of generations expected for failed runs preceding a successful run. Two different formulas were developed. One (section 7.2.5) uses the data to project a mean for the entire population of seeds. The rankings and speedups based on this formula appear in figure 8.69. The other formula for $G\mu$ (section 7.2.9) represents the mean

for the set of seeds used to generate the data. The data based on this formula appear in figure 8.70.

Comparing the results for the two $G\mu$ s (figures 8.69 and 8.70), the rank order on all lists are identical. The speedups in figure 8.69 tend to be slightly higher than in figure 8.70. Based on this, the two $G\mu$ s are equivalent predictors of performance. Comparing both to figure 8.68 in which the failure rate and successful runs are combined in a different way than for the $G\mu$ s, there are differences in only five pairs. Two reversals of rank order appear in the last four methods of the list for 32K, 100 seed, $maxgen=100$. Two pairs (B2 and S2, and M1 and M2) are also reversed on the 32K, 100 seed, $maxgen=75$ list. The last difference affects B2 and S1 on the 64K, 30 seed, $maxgen=70$ list. In each of these cases the two members of the inverted pair have comparatively close performance. In all three figures (8.68 through 8.70) M1 and M2 top all the lists, and except for the 32K experiments, M2's performance surpasses that of M1. In all but one case (figure 8.70, 32K, 30 seed, $maxgen = 100$, M2), the *U Test* supports the superiority of M1 and M2 over B1 with a high degree of confidence.

16K. 30 seed $maxgen = 125$	16K. 30 seed $maxgen = 100$	16K. 100 seed $maxgen = 125$	16k. 100 seed $maxgen = 100$
M2 2.58	M2 2.85	M2 1.79	M2 2.01
M1 1.97	M1 2.21	M1 1.39	M1 1.49
B2 1.19	B2 1.12	S1 1.03	S1 1.03
B1	B1	B1	B1
S2	S2	B2	S2
S1	S1	S2	B2

Figure 8.69: Summary for ranking by $G\mu$ using the formula for the population, with speedup over B1. Data condensed from figures 8.20, 8.30, 8.38, 8.46, 8.54, and 8.62. Continued on next page.

32K, 30 seed <i>maxgen</i> = 100	32K, 30 seed <i>maxgen</i> = 75	32k, 100 seed <i>maxgen</i> = 100	32K, 100 seed <i>maxgen</i> = 75
M1 1.78	M1 2.20	M2 1.85	M2 2.23
M2 1.13	M2 1.39	M1 1.70	M1 2.20
B2 1.05	B1	B2 1.25	S1 1.25
B1	B2	S1 1.23	B2 1.14
S2	S2	B1	S2 1.06
S1	S1	S2	B1
64K, 30 seed <i>maxgen</i> = 70	64K, 30 seed <i>maxgen</i> = 60	64K, 30 seed <i>maxgen</i> = 50	
M2 2.68	M2 3.73	M2 4.29	
M1 1.94	M1 2.76	M1 3.31	
B2 1.51	S1 1.76	S1 1.37	
S1 1.39	B2 1.70	S2 1.17	
S2 1.18	S2 1.48	B2 1.01	
B1	B1	B1	
64K, 100 seed <i>maxgen</i> = 70	64K, 100 seed <i>maxgen</i> = 60	64K, 100 seed <i>maxgen</i> = 50	
M2 1.89	M2 2.30	M2 5.18	
M1 1.54	M1 1.91	M1 4.58	
S2 1.12	S2 1.25	S2 2.25	
S1 1.11	S1 1.22	S1 1.90	
B1	B2 1.07	B1	
B2	B1	B2	

Figure 8.69 **continued**: Summary for ranking by $G\mu$ using the formula for the population, with speedup over B1. Data condensed from figures 8.20, 8.30, 8.38, 8.46, 8.54, and 8.62.

16K, 30 seed <i>maxgen</i> = 125	16K, 30 seed <i>maxgen</i> = 100	16K, 30 seed <i>maxgen</i> = 125	16K, 30 seed <i>maxgen</i> = 100
M2 2.49 .999	M2 2.71 .999	M2 1.78 .999	M2 1.99 .999
M1 1.92 .999	M1 2.13 .999	M1 1.39 .999	M1 1.48 .999
B2 1.18 .99	B2 1.12	S1 1.03	S1 1.03
B1	B1	B1	B1
S2	S2	B2	S2
S1	S1	S2	B2
32K, 30 seed <i>maxgen</i> = 100	32K, 30 seed <i>maxgen</i> = 75	32K, 100 seed <i>maxgen</i> = 100	32K, 100 seed <i>maxgen</i> = 75
M1 1.76 .999	M1 2.14 .999	M2 1.84 .999	M2 2.20 .999
M2 1.14	M2 1.38 .999	M1 1.69 .999	M1 2.17 .999
B2 1.05	B1	B2 1.24 .999	S1 1.25 .999
B1	B2	S1 1.23 .999	B2 1.13 .96
S2	S2	B1	S2 1.06 .999
S1	S1	S2	B1

Figure 8.70: Summary for ranking by $G\mu$ using the formula for the sample, with speedup over B1. Data condensed from figures 8.22, 8.31, 8.39, 8.47, 8.55, and 8.63. The second number on each list is the degree of confidence that the performance of the given method is better than B1 when the degree of confidence is $\geq .95$. **Continued on next page.**

64K, 30 seed <i>maxgen</i> = 70	64K, 30 seed <i>maxgen</i> = 60	64K, 30 seed <i>maxgen</i> = 50
M2 2.59 .999	M2 3.48 .999	M2 3.84 .999
M1 1.89 .999	M1 2.60 .999	M1 3.00 .999
B2 1.48 .999	S1 1.69 .999	S1 1.32
S1 1.37 .999	B2 1.63 .999	S2 1.15
S2 1.17 .99	S2 1.43 .999	B2 1.01
B1	B1	B1
64K, 100 seed <i>maxgen</i> = 70	64K, 100 seed <i>maxgen</i> = 60	64K, 100 seed <i>maxgen</i> = 50
M2 1.88 .999	M2 2.28 .999	M2 4.93 .999
M1 1.54 .999	M1 1.89 .999	M1 4.36 .999
S2 1.12 .999	S2 1.25 .999	S2 2.17 .999
S1 1.11 .999	S1 1.21 .999	S1 1.84 .999
B1	B2 1.06 .99	B1
B2	B1	B2

Figure 8.70 **continued**: Summary for ranking by $G\mu$ using the formula for the sample, with speedup over B1. Data condensed from figures 8.22, 8.31, 8.39, 8.47, 8.55, and 8.63. The second number on each list is the degree of confidence that the performance of the given method is better than B1 when the degree of confidence is $\geq .95$.

The consistent trends appearing in the data analyzed in sections 8.1 and 8.2 are discussed in the next section.

8.3 Evaluation summary

Definite consistent patterns appear, from examining the data and considering this analysis both in terms of evolutionary progress (section 8.1) and efficiency (section 8.2). In both evaluations, the SFS algorithm with or without competition does not exhibit any consistent advantage over the benchmarks; at times it finds results sooner, at times later than the benchmarks. Because implementing SFS in a genetic algorithm means that the GA does additional work on each member of the parent pool, it does not seem practical to add this algorithm to a GA which is being designed to find a solution to a problem.

Each new operation (MTF, SFS, competition) adds additional processing to the benchmark as opposed to simply replacing one operation with another. Consequently, the research was undertaken to determine whether any gains would be made in the number of generations needed to find a solution, as a result of adding any of these algorithms to the benchmark; but if such gains did manifest, there was a concern as to whether these gains would be lost timewise because the extra work requires extra processor time. Based on the data, improvements in performance over the benchmark tend to be less dramatic when based on processor time than when based on the number of generations. The available timing data indicate that the time factor sometimes alters the rank order, but only when two methods have close performance based on the number of generations needed to get a solution. In these cases, while the rank order may be different, the performance based on processor time is still close.

All variations of the MTF algorithm generally seem to speed up the evolutionary process, but none of the $M1-i$ and $M2-i$, $i \in \{2,3,4,5\}$, consistently top $M1$ and $M2$. The purpose of applying MTF every i^{th} generation (as opposed to every generation) is to save time. Since there is no consistency in the performance of the $M1-i$ and $M2-i$ and since the timing did not change this drastically, the focus was moved to $M1$ and $M2$.

$M1$ and $M2$ consistently perform better than the benchmark in the analysis of the evolutionary process (section 8.1.2) and almost consistently in the analysis of the efficiency (section 8.2.2). $M2$ has a slight advantage over $M1$. $M1$ lacks competition,

does less work, and consequently is slightly easier to implement than M2. M1 is consistently better than the benchmark, but based on some measures, only marginally better. On the other hand, based on the different criteria for efficiency, M2 performs better than M1 more frequently, at times showing far superior performance, but in one instance (figure 8.26, $maxgen=100$) M2 just matches the performance of B1. The advantage of these two MTFs is enhanced by the fact that they locate smaller FSMs.

The superiority of these two methods varies greatly. When looking at the evolutionary process (section 8.1.2) both clearly outperform the benchmark and SFSs with M2 having a slight edge over M1 (figures 8.5 and 8.6). When looking at convergence the superiority of M1 and M2 over the benchmarks and SFSs is evident, except in one instance (figure 8.26, $maxgen=100$). This advantage appears across the different criteria, across the six experiments, and across the different values of $maxgen$ within each experiment. When applicable, the *U Test* indicates that the difference in the performance of M1 and M2 with respect to the benchmark is indeed significant across the different experiments, criteria, and $maxgens$ except in two instances (16K/100 seed/ $G_{AV}/maxgen=100/M2$ and 32K/30 seed/probability of getting a solution first based on number of generations/ $maxgen=100/M2$). In all but these two instances, the degree of confidence that the difference in performance is not due to chance exceeds .95.

The three criteria which combine success and failure (probability of finding a solution first and the two $G\mu s$) yield similar results when compared to each other.

Therefore any one of these three criteria should suffice as a measure of performance. The formula for $G\mu$ as a predictor of population performance is the easiest to implement, but for the other two criteria the *U Test*, can be used to determine the degree of confidence that the difference in the performance of two methods is significant. Since the same data points are used for $G\mu$ for the sample and the percent increase of getting a solution first, the degrees of confidence for these two criteria will be identical.

The criteria which include the failure rate are considered better predictors of real performance than those which ignore the failure rate. The difference between two successful runs is at most *maxgen*; usually it is much less. The difference between a single failed run followed by a successful run and say four failed runs followed by a successful run is $4(\text{maxgen}+1)+G_{AV} - [(\text{maxgen}+1)+G_{AV}] = 3(\text{maxgen}+1)$. Thus, when looking for a solution to a given problem, the failure rate is critical to the number of generations expected to be bred or amount of processor time expected to be used in order for the GA to locate a solution. The strong impact of the number of failures on the values of $G\mu$ and $T\mu$ regardless of which set of formulas is used, and the fact that K (section 7.2.2.10) only considers the failure/success rate supports the argument that the failure rate is an important factor. The advantage of K as opposed to the other criteria is that it includes a probabilistic model with a degree of certainty in the formula, and it is easy to implement. When the probability of getting a solution first is the criterion, increases in performance far in excess of 100% are not uncommon. The *U Test* generally supports the conclusions

based on the probability of getting a solution first with a high degree of certainty.

Based on the data evaluated in this chapter, the inclusion of MTF, with or without competition, seems to show enough benefit to be included in a GA with an FSM genome, but this conclusion is based on experiments on a single problem and it was felt that additional testing was needed. The next chapter details the additional testing to a new problem set and its evaluation.

>>>●<<<

Chapter 9

Performance across a new problem set

As shown in chapter 8, testing indicated that the MTF algorithm resulted in faster (fewer generations and less processor time) convergence to a solution. The boost that MTF gave the GA on the trail following problem is impressive, but a set of tests on a single problem is not sufficient. The question arises as to whether the results are particular to this problem or whether the results will carry across other problems.

The problem considered in this chapter is that of language recognition. The Tomita Test Set (1982) was used for further testing. This set consists of 14 languages. It was augmented with 6 additional languages. During experimentation, the application of the GA to two of these 20 languages was deemed too trivial to get useful data. The 18 remaining languages were then used for the experiments to examine the effect of SFS and MTF on GA efficiency/convergence. The next section details the additional experiments undertaken to see if the results of chapter 8 would be replicated. Then section 9.2 contains the evaluation criteria applied to the data from these experiments, and in section 9.3 the data from these new experiments are evaluated.

9.1 The experimentation

A standard testbed was needed to further test the effect of MTF and SFS on convergence. The Tomita Test Set (1982) consists of sets of strings representing 14 regular languages. Tomita used these sets to breed finite state automata (FSA) language recognizers. Angeline (1996) wrote that the Tomita Test Set is a semi-standard test set used to "induce FSAs [finite state automata] with recurrent NNs [neural networks]." Consequently, the Tomita sets (1982) representing seven pairs of complementary languages were used for testing. In addition, Angeline (1996) further "suggest[ed] that a better test set would include some languages that are not regular. . . . Something beyond the capability of the representation so you can see how it tries to compensate." As per Angeline's suggestion, three additional sets were drawn up. These three sets were designed to represent languages which are not regular; however, reducing the description of each of these languages to a finite set as Tomita did, effectively reduces each language to a regular language. When creating each set for the three additional languages, an attempt was made to capture some properties of the strings in the language. The Tomita test set and three additional languages are presented in section 9.1.1.

9.1.1 The languages

There are seven Tomita sets (1982). Each set represents two languages and consists of two lists: a list of strings belonging to one of two of the defined regular languages and a list of strings which do not belong to that language. In the lists

which follow, λ represents the empty string. Seven of the Tomita languages and sets representing those languages are as follows:

L1: 1^*

<u>in L1</u>	<u>not in L1</u>
λ	0
1	10
11	01
111	00
1111	011
11111	110
111111	1111110
1111111	1011111
11111111	

L2: $(10)^*$

<u>in L2</u>	<u>not in L2</u>
λ	1
10	0
1010	11
101010	00
10101010	01
10101010101010	101
	100
	1001010
	10110
	110101010

L3: Any string which does not contain an odd number of consecutive zeroes at some point in the string after the appearance of an odd number of consecutive ones.

<u>in L3</u>			<u>not in L3</u>		
λ	11	111	10	1110	1001000
1	00	000	101	1011	11111000
0	100	100100	010	10001	0111001101
01	110	110000011100001	1010	111010	11011100110
		111101100010011100			

L4: No more than two consecutive 0's

<u>in L4</u>	<u>not in L4</u>
λ	000
1	11000
0	0001
10	00000000
01	11111000011
00	1101010000010111
100100	1010010001
001111110100	0000
0100100100	00000
11100	
010	

L5: Even length strings which, when the bits are paired, have an even number of 01 or 10 pairs

<u>in L5</u>	<u>not in L5</u>
λ	1
11	0
00	111
1001	010
0101	00000000
1010	1000
1000111101	01
1001100001111010	10
111111	1110010100
0000	010111111110
	0001
	011

L6: The difference between the number of 1's and 0's is 3n.

<u>in L6</u>	<u>not in L6</u>
λ	1
10	0
01	11
1100	00
101010	101
111	011
000000	11001
10111	1111
0111101111	00000000
100100100	010111
	10111101111
	1001001001

L7: $0^*1^*0^*1^*$

<u>in L7</u>	<u>not in L7</u>
λ	1010
1	00110011000
0	0101010101
10	1011010
01	10101
11111	010100
000	101001
00110011	100100110101
0101	
0000100001111	
00100	
011111011111	
00	

The two lists of strings within each of these languages are interchanged to define seven more languages labelled L1c through L7c.

Taking Angeline's suggestion (1996), three representative sets for three non-regular languages were drawn up. They are as follows:

L8: prime numbers in binary form

<u>in L8</u>			<u>not in L8</u>
10	0000011	λ	0110
11	0000000011	0	0000110
101		1	0000000110
111		100	110011
1011		110	10110
1101		1000	11010
11111		1001	111110
100101		1010	10010110
1100111		1100	10010
10010111		1110	1001011
011		1111	111111
0011		11001	

L9: $0^i 1^i$ **in L9**

λ
 01
 0011
 000111
 00001111
 0000000011111111

not in L9

0	010	1010
1	011	1100
00	100	1010011000111
10	101	010011000111
11	110	010101
000	111	01011
001	1000	

L10: Binary form of perfect squares**in L10**

0	0000100
1	0000000100
100	10000
1001	100000000
11001	1100100
0100	11001000000

not in L10

λ	1110000
10	111000000
11	11100000000
101	00000111
110	000000111
111	100000
1100	100000000000
00111	1100100000

There were several considerations which were behind the selection of the strings in the sets for L8, L9, and L10. When designing these sets,

- 1) an attempt was made to keep the sets as small as possible even though a larger set would provide a better representation. Already the sets for L8, L9, and L10 are larger than any of the Tomita's sets (1982).
- 2) The sets should include some strings indicating a "pattern" in a language. For example preceding a binary number with a string of zeroes does not change its value; therefore in L8 and L10 a string preceded by zeroes is on the same list as the shorter string. Similarly appending "00" to the end of a binary number effectively multiplies it by four; consequently in L10 a string ending with an even number of zeroes belongs on the same list as the one without these trailing zeroes. A few

representative strings for each of these "patterns" have been included for L8 and L10.

In addition to defining L1 through L10, these ten sets are used to define ten more languages labelled L1c through L10c; by interchanging the list of strings belonging to a language with the list of strings which do not belong to that language another ten languages (L1c through L10c) are defined. The languages L_i and L_{ic} are called complementary languages.

The next section will look at some aspects of the problem of accepting or rejecting strings in a language.

9.1.2 Algorithmic and programming considerations

While the solution to the language acceptance problem can be modelled by a finite state automaton, it differs significantly from the trail problem. First, using the GA to find an FSA for L1 and L1c was deemed too simple a problem to provide useful data for this study. Amongst other issues, finding a set of 100 seeds such that none of the seeds would result in a solution in generation 0 proved to be difficult.

Tomita (1982) focussed on finding an FSA corresponding to the regular expression or verbal description of the language. In the research for this thesis, no attempt was made to find complete solutions to the original verbal descriptions or regular expressions defining the languages. Rather, the work for this thesis was restricted to finding an FSA which defined the membership function for the representative set of strings for a given language. For any given language the fitness function tested the strings on the two lists for that language. The fitness of an

individual is the total number of strings which are correctly identified as to their membership or lack of membership in the language.

The benchmark, SFS, and MTF, each with and without competition, were applied to a total of 18 sets—nine of the original ten sets (L2 through L10) and their complements (L2c through L10c). As the data became available it was clear that MTF was not necessarily predominant with respect to efficiency. Consequently, two hybrid methods were added to these experiments to see if either would consistently perform better than B1. For these hybrids, MTF and SFS were applied to alternating generations, both with and without competition. MTF was applied to all even generations including to the initial generation, and SFS was applied to every odd generation. These two hybrids were labelled A1 (no competition) and A2 (competition included).

Several modifications were made to the programs used for the trail problem. Obviously the fitness function had to be changed for each language. The mapping of the FSA table into the genome for the language problems is different from that for the trail problem. Consequently, the genome had to be modified, and the program functions involved in genotype to table conversion and vice versa also had to be changed. Based on the nature of the trail problem, the finite state machine (generic FSM defined in section 4.1) used to model a solution for the trail following problem is a transducer which produces an output/reaction each time a transition rule is applied. Thus the table for the trail problem requires an output function in addition to the transition function. In addition the set of final states for the trail

problem is empty.

In the case of the language recognition problem, an FSA is used to examine strings to determine whether or not they belong to a language. The finite state automaton implemented for this part of the study does not give a response with each application of a transition rule. Thus, an output function per se is not defined. Instead, when applying an FSA to a string and when the last character of a string brings the FSA into a final state, the string is accepted as a member of the language; otherwise the string is rejected. Hence, in order to define a string's membership in a language, the FSA's set of final states cannot be empty.

The FSA used to identify the membership function of strings with respect to a given language requires that each present state be identified as to whether it is or is not a final state. This requires a single bit; 1 is used to indicate a final state. The 18 languages have two characters in their alphabets. Therefore for each present state two positions are needed in the genome for next states. The number of bits needed for each future state is strictly dependent on the maximum number of states permitted.

There were two lines of thought as to how to order the three pieces of data for each present state. Since the single bit defining a state as a final state is associated with the present state it comes first. It is followed by the next state for an input of 0, which is then followed by the next state for an input of 1. This will be referred to as genome G1. However, within G1 this ordering of the three pieces of data for a state actually ties that single bit closer to the next state for input 0 with

respect to the crossover operator. In this case, a positive even number of crossover points are needed between the single bit defining final state status and the next state for input 1 in order to transfer that single final state bit to a child with the next state for an input of 1 and without the next state for input 0. Lemmas 9.2 and 9.3 indicate the probability bias against sending the final state bit with only the next state for input 1 (lemma 9.2) to a child, as opposed to sending the final state bit with only the next state for input 0 (lemma 9.3) to a child.

For the lemmas of this chapter, let p_x be the per bit probability of crossover and denote by M is the number of bits used to represent the state number. Then there are $M-1$ possible crossover points within a state number. In addition, for the next state for input 0 two additional crossover points must be considered: one between the bit defining final state status and the next state for input 0, and the other between the two next states. Hence, there are a total of $M-1+2 = M+1$ possible crossover points relevant to the next state for input 0, and there are $\left\lfloor \frac{M+1}{2} \right\rfloor$ positive even integers $\leq M+1$.

The following lemma is needed for the proofs which follow:

Lemma 9.1: $\sum_{i=1}^M (1-p_x)^{i-1} p_x = 1 - (1-p_x)^M.$

Proof: For a geometric progression $\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}$. Therefore,

$$\sum_{i=1}^M (1-p_x)^{i-1} p_x = p_x \sum_{i=0}^{M-1} (1-p_x)^i = p_x \frac{1-(1-p_x)^M}{1-(1-p_x)} = 1 - (1-p_x)^M. \quad \square$$

The next two lemmas are used to show the internal bias in G1 with respect to crossover and the final state bit.

Lemma 9.2: When breeding an offspring, the probability of crossover retaining both

the single bit defining final state status and the next state for an input of 1, and of disrupting the next state for input 0 is

$$(1-p_x)^{2M} \sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} \left(\frac{p_x}{1-p_x} \right)^{2i} C_{2i}^{M+1}.$$

Proof: The probability of crossover retaining both the single bit defining final state status and the next state for an input of 1, and of disrupting the next state for input zero

= Σ (the positive number of ways to choose an even number of distinct crossover points within the $M+1$ possible crossover points relevant to the next state for input 0)
 $\times$ (probability of getting that even number of crossover points, and also having no crossover points within the bits for the next state for input 1)

= Σ [(the positive number of ways to choose an even number of distinct crossover points within the $M+1$ possible crossover points relevant to the next state for input 0)
 $\times$ (probability of getting that even number of crossover points)
 $\times$ (probability of having no crossover points within the bits for the next state for input 1)]

= $\sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} C_{2i}^{M+1} \times p(2i \text{ crossover points within the } M+1 \text{ crossover points relevant to the next state for input 0 and no crossover points within the remaining } M+1-2i \text{ positions relevant to the next state for input 0})$
 $\times p(M-1 \text{ points without crossover for the next state for input 1})$

$$= \sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} C_{2i}^{M+1} p_x^{2i} (1-p_x)^{M-2i-1} (1-p_x)^{M-1} = \sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} p_x^{2i} (1-p_x)^{2(M-i)} C_{2i}^{M+1}$$

$$= \sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} p_x^{2i} \frac{(1-p_x)^{2M}}{(1-p_x)^{2i}} C_{2i}^{M+1} = (1-p_x)^{2M} \sum_{i=1}^{\left\lfloor \frac{M+1}{2} \right\rfloor} \left(\frac{p_x}{1-p_x} \right)^{2i} C_{2i}^{M+1}.$$

This formula assumes that at least one bit within the next state for input 0 will be

replaced by crossover, and that no crossover point will occur within the next state for input 1. $\square$

For all but two of the languages (L2, and L2c) used for this part of the study. $M=4$; that is the genome allows for 16 states. p_x is .01 as it was for the trail problem. Thus for a given present state, the probability that the single bit defining membership in the set of final states will be sent to a child with only the next state for input 1 is $.99^8 \sum_{i=1}^2 \left(\frac{.01}{.99}\right)^{2i} C_{2i}^5 = .99^8 \left[\left(\frac{.01}{.99}\right)^2 C_2^5 + \left(\frac{.01}{.99}\right)^4 C_4^5 \right] \approx 9.4 \times 10^{-4}$.

Lemma 9.3: For a given present state the probability that the single bit defining membership in the set of final states will be sent to a child with only the next state for input 0 intact is $(1-p_x)^M [1 - (1-p_x)^M]$.

Proof: The probability that the single bit defining membership in the set of final states will be sent to a child with only the next state for input 0 intact

= p(no crossover point between the single bit and the next state for input 0, and no crossover point within the bits of the next state for input 0, and at least one crossover point either between the two next states or within the next state for input 1)

= p(no crossover point between the single bit and the next state for input 0) $\times$ p(no crossover point within the bits of the next state for input 0) $\times$ p(at least one crossover point either between the two next states or within the next state for input 1)

$$= (1-p_x) (1-p_x)^{M-1} \sum_{i=1}^M [(1-p_x)^{i-1} p_x].$$

Applying lemma 9.1 and simplifying yields $(1-p_x)^M [1 - (1-p_x)^M]$. $\square$

The value of this probability for a 16-state machine ($M=4$) and $p_x=.01$ is approximately .04. This shows that for a given present state the single bit defining membership in the set of final states is much more likely to transfer to a child with

only the next state for input 0 intact than it is to transfer to a child with only the next state for input 1 intact.

If instead the bit defining membership in the set of final states is positioned between the two future states for a present state, call this genome G2. crossover is equally likely to send that bit to a child with either of the next states. To avoid the bias indicated in the previous paragraph the experiments were carried out with genome G2.

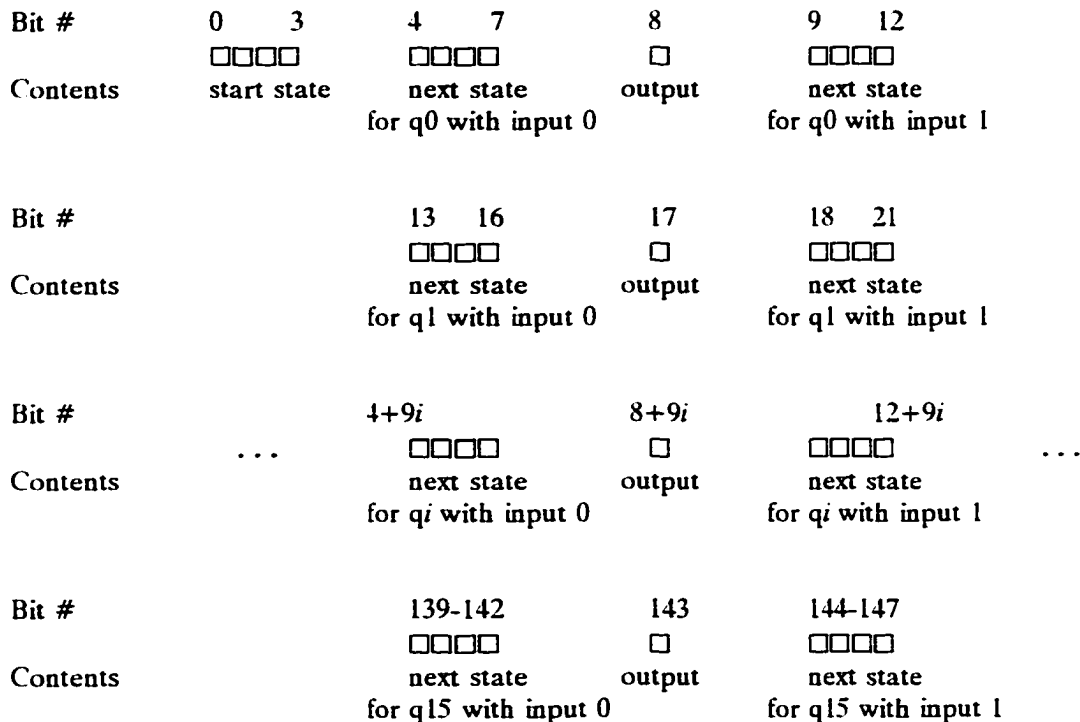


Figure 9.1: 16-state/148-bit FSA Genome Map: bit labelled as output defines final state status.

With respect to the mapping of the FSA table into the genome for G2, each line of the transition table of the FSA contains two next states and the membership

function with respect to membership in the set of final states. As in the case of the trail problem's genotype, the start state forms the first field in the genome. Then the lines of the transition table are listed sequentially in the genome. As previously indicated, the bit defining membership in the set of final states is positioned between the two future states for each present state. Figure 9.1 shows the layout for such a genome allowing for a 16-state FSA.

Tomita (1982) presented a set of solutions for his language recognition problems. Some of these solutions were in minimal form, consisting of a minimum number of states. The maximum number of states he presented for a single language was six. Clearly it was not necessary to allow for a 32-state solution. Thus, genome sizes of 4, 8, and 16 states were tried and population sizes of 2^4 up to 2^{10} were tried depending on the size of the genome. (All population sizes were powers of 2.) Note that a smaller genome has a smaller genome space, therefore a smaller population size can be used. The parent pool used to breed the next generation retained the the top 5% of the current population. Also, the parent pool contained a minimum of two parents in order to permit the possibility of crossover between two different parents (as opposed to crossover between copies of a single individual).

In an attempt to choose appropriate values for the GA parameters, it was noticed that a search by the GA for FSAs to define L1 and L1c was inappropriate to provide useful data for this study due to the simplicity of these problems. For several of the languages the GA found a solution in generation 0. The parameters were selected in order to minimize this situation. Table 9.1 indicates the final set of

parameters chosen for each of the remaining languages. For the languages with a maximum of eight states (L2 and L2c) the genome contains 3 bits for the start state + 8 states $\times$ (2 inputs $\times$ 3 bits for each next state + 1 bit for final state status) = 59 bits. Similarly for the languages with a maximum of 16 states, the number of bits in the genome is $4 + 16(2 \times 4 + 1) = 148$ bits. **Figure 9.1** (which appears on page 290 as opposed to **table 9.1** which follows) contains the 16-state genome map.

Language	Maximum # of states allowed	Population size	Size of parent pool
L2 and L2c	8	32	2
L3 and L3c	16	128	6
L4 and L4c	16	64	3
L5 and L5c	16	128	6
L6	16	64	4
L6c	16	64	3
L7, L7c, L8, L8c, L9, and L9c	16	128	6
L10 and L10c	16	256	13

Table 9.1: Parameters for the Languages

The set of 100 seeds from the trail problem was used with a few changes for L2 and L2c. A few of the seeds for each of these languages yielded a solution in the initial generation. Since the GA terminates when a solution is found, the few seeds which resulted in a solution in generation 0 were changed to permit the GA to move beyond the initial population in every run.

The runs were permitted to breed a maximum of 1000 generations, recording the per run data as indicated in figure 7.1 of section 7.1. To see if the results would carry across a wide range of *maxgens* (maximum number of generations bred) rather than across only a few good choices for *maxgen*, data were collected for a wide range of values from unrealistically low to 1000, which is extremely high. This is in consideration of the fact that a researcher using a GA to locate the solution to a problem could very easily select an inappropriate value for *maxgen*. Hence data were recorded for *maxgens* of 1000, 750, 500, 300, 250, 200, 150, 100, 75, 50, and 25.

The next section details the criteria used to evaluate the data.

9.2 Evaluation criteria

The 18 experiments (L2-L10 and L2c-L10c of section 9.1.1) with 8 methods per experiment (B1, B2, M1, M2, S1, S2, A1, A2), 100 runs per method, and 11 values of *maxgen* per run generated an immense amount of data. The data were subject to nine criteria for evaluation of efficiency. The criteria are K (section 7.3.10.1), G_{AV} (average number of generations for a successful run), $T_{AV:S}$ (processor time corresponding to G_{AV}), the two $G_{\mu s}$ and $T_{\mu s}$ (sections 7.3.5 and 7.3.7 for the population of all possible seeds, and section 7.3.9 for the sample), and the probability of getting a solution first based on the number of generations and the corresponding processor times. The speedup or percent increase when applicable (section 7.3.10.1.) was summarized for the nine criteria. The percent decrease for average machine size was also summarized and examined, along with the *U Test*

degree of confidence that each method is better than B1 with respect to average machine size. These summaries appear on the worksheets which can be found in appendix D.

A run with 100 seeds did not take too long: several minutes to under two hours real time per run as opposed to six to 25 days for the trail problem. Available resources permitted collection of reliable timing data. Thus, timing data are available for all the experiments.

On the worksheets, the methods are recommended on two levels: first on the criterion level, then on the language level. The worksheets containing these recommendations appear in appendix D and are analyzed in the next section. The worksheet/summary for each language consists of two pages. The box at the top left of the first sheet contains the methods recommended for the language. These recommendations are based on a language showing some improvement in efficiency over B1 and rarely if ever being less efficient than B1. On the summary for each language there are two tables for average machine size on the right side of the first sheet (one for the percent decrease from B1 and the other for the *U Test* results), and a separate table for each of seven of the nine criteria (subsequent tables). The tables for $G\mu$ and $T\mu$ for the sample have been omitted from the summary because in most cases they are identical to the tables for $G\mu$ and $T\mu$ for the population. When there are differences in these tables, the differences are minimal and have no effect on the recommendations.

Also appearing in the box at the top left of the first sheet are a range of

values for F and G_{AV} listed as "lo" and "hi." Essentially the intervals indicate the range for the lowest and highest values for F and G_{AV} across the methods. For F the first number of each interval is the lowest/highest value of F for $maxgen=1000$; the larger number of each interval is the lowest/highest value of F for $maxgen=25$. This is because there are fewer failures when $maxgen$ is larger. For G_{AV} the lower values appear when $maxgen=25$, and the higher numbers occur at $maxgen=1000$. This is due to the fact that G_{AV} decreases with $maxgen$ since the numbers of generations that are larger than the modified value of $maxgen$ are dropped from the average as $maxgen$ is reduced.

To the left of some methods are the symbols blank, ?, s, or *. These characters are the criterion level recommendations. The blank indicates that the method is not as efficient as B1 for the given criterion. The question mark indicates that it is not conclusive how to rate a method. The s indicates that the method is considered equivalent in performance to B1. The * indicates recommendation because the method is considered to have better performance than B1. The appearance of two of the non-blank symbols for a method indicates a range of possible conclusions. A method is recommended for a criterion if the method generally shows improvement over the benchmark across most of the eleven $maxgens$, occasionally (if ever) matches the performance of the benchmark, and rarely (if ever) falls below the benchmark in a ranking. Poor performance for the lowest and/or highest values of $maxgen$ is not considered a deterrent to recommendation.

A method is recommended for a language based on the following four

considerations:

- 1) It is marked with at least one of the three symbols on each of the tables for the nine criteria.
- 2) Most of the markings are *.
- 3) The timing criteria are considered more important than the criteria based on the number of generations.
- 4) G_{AV} and $T_{AV:S}$ are not considered critical to the recommendation except when there are very low failure rates.

To elaborate with an example of consideration 4 above, for L8c almost all of the methods do poorly with respect to G_{AV} and $T_{AV:S}$ with smaller values of *maxgen*. The failure rate is high for the corresponding values of *maxgen*. As explained in section 8.3, a high failure rate dominates the values (except for G_{AV} and $T_{AV:S}$) for the number of generations and amount of processor time when considering efficiency. Consequently, the poor performance of the methods with respect to G_{AV} and $T_{AV:S}$ for the smaller values of *maxgen* does not prevent recommendation of a method for L8c.

Essentially a method is recommended for a language if it *performs well* for the language. *Performs well* is interpreted to mean that the method is recommended for most but not necessarily all criteria.

The nine criteria (K , G_{AV} , $T_{AV:S}$, the two $G_{\mu s}$ and $T_{\mu s}$, and the probability of getting a solution first based on the number of generations and the corresponding processor times) are used to determine the efficiency of the methods as compared to B1. Also of interest is the effect of the methods on machine size. The percent decrease in machine size with respect to B1 and the *U Test* degree of confidence that

a method is better than B1 are summarized and the methods are ranked based on the range of the percent decreases across the eleven values of *maxgen* and a high degree of confidence from the *U Test*. When the range for two methods is similar, the values for each *maxgen* are examined to determine rank.

The next section presents the evaluation of these experiments based on the above described criteria.

9.3 Evaluation

The nine criteria selected in the previous section to evaluate efficiency and the single criteria for machine size are applied to 18 experiments: nine languages (L2 through L10) and the nine complementary languages (L2c through L10c). Each experiment consists of eight methods:

- 1) B (basic GA or benchmark), M (MTF), S (SFS), and A (a hybrid which alternates MTF and SFS).
- 2) The letter is followed by a 1 (no competition) or 2 (competition incorporated into the fitness function).

Since machine size is a separate consideration, it is discussed here first. With respect to machine size the rankings only consider those machines which generally reduce machine size by more than three percent (empirically chosen) when compared to B1 across the eleven values of *maxgen*. Those which perform similar to or worse than B1 are left out. (See worksheets in appendix D.) In addition a *U Test* was used to compare each method to B1. Any method which generally (across the 11 *maxgens*) has a degree of confidence $< .90$ is also omitted. The rankings are

determined by the range of the percent decreases for the average machine size as compared to B1 across the 11 *maxgens*. When the ranges of the percent decreases are similar for two methods, the specific values across the lines on the table in appendix D are then compared. When two methods appear on the same line of a list, they are considered to be equally effective in locating solutions with fewer states than B1. The rankings are as follows:

<u>L2</u>	<u>L2c</u>	<u>L3</u>	<u>L3c</u>	<u>L4</u>	<u>L4c</u>
M2	A1	M1, M2	M2	M2	M2
A2	M1	A1, A2	M1	A2	M1
A1	M2		A1	M1	A2
M1			A2	A1	S1
S1					
<u>L5 & L10</u>	<u>L5c</u>	<u>L6</u>	<u>L6c</u>	<u>L7</u>	<u>L7c</u>
M2	M2	M2	A2	M2	M1
A1	M1	A1, A2	M1	M1	M2
M1		M1	M2	A1	A2
					A1
<u>L8</u>	<u>L8c</u>	<u>L9</u>	<u>L9c</u>	<u>L10 & L5</u>	<u>L10c</u>
M1, M2	M2	M1, M2	M1	M2	M1
A1, A2	M1	A2		A1	M2
	A1, A2	A1		M1	A1
	S1, S2, B2				A2
					S1.

Note that M1 and M2 produce smaller FSAs than B1 as indicated by the fact that both methods are on all the lists except one. For L9c, M2 did produce smaller FSAs than B1, but not enough to make the list. (As mentioned above, the methods appearing on the above lists are placed there only if they produce FSMs which are generally more than 3% smaller than those produced by B1 across the eleven *maxgens*.) In addition, the *U Test* did not provide a high degree of confidence ($\geq .90$) that M2 was better than B1. Also note that A1 and A2 appear on the lists rather

frequently indicating that the MTF part of these hybrids tends to influence these hybrids toward smaller solutions, just not as consistently as MTF on its own.

With respect to efficiency/convergence, the results are first presented with an eye to the methods recommended for each language. The methods recommended for each language are as follows (order does not indicate that one method is better than another):

L2, L6: M1, M2, and A1 are recommended.

L2c: M1 and A1 are recommended.

L3: S1 and S2 are recommended.

L3c, L10: S1 is recommended.

L4: S1, S2, and A2 are recommended.

L4c: S1, S2, and A1 are recommended.

L5, L5c, L7: No method stands out as being consistently better than B1.

L6c: M1, A1, A2, and S1 are recommended.

L7c: M2 is recommended.

L8: S1 and B2 are recommended.

L8c: S1, S2, A1, and A2 are recommended. All do poorly on G_{AV} and $T_{AV:S}$ with smaller values of *maxgen* (See appendix D.), but the failure rate is high for these values of *maxgen* and a high failure rate has more of an influence on the total number of generations and amount of processor time.

L9, L9c: B1 is the most efficient.

L10c: S1 is good for $\text{maxgen} \geq 200$ based on criteria which include the failure rate.

The same recommendations presented by method are as follows:

M1: L2, L2c, L6, L6c;

M2: L2, L6, L7c;

A1: L2, L2c, L4c, L6, L6c, L8c;

A2: L4, L6c, L8c;

S1: L3, L3c, L4, L4c, L6c, L8, L8c, L10;

S2: L3, L4, L4c, L8c;

B1: L9, L9c; and

B2: L8.

Note that the methods with competition do not seem to do as well as those without competition. Each is recommended for fewer languages than the corresponding

method without competition.

Clearly no one method prevails based on this data. S1 is recommended more than the other methods (8 times out of 18 possibilities); this is not enough for it to be considered generally better than B1 since B1 either matches or excels S1 for the remaining 10 experiments. This is the same conclusion that was made from the experiments with the trail problem. The big surprise is that MTF did not fare as well as it did for the trail problem. This phenomenon is analyzed in the next section.

9.4 Performance of MTF

The analysis of the data for the trail problem (chapter 8) indicates that MTF outperforms the other methods. The summary of the method recommendations for the languages in the previous section does not support the conclusions obtained from the trail problem (section 8.3) with respect to the efficiency of MTF. To understand why MTF performed so well on the trail problem and failed to do so for the languages it is necessary to look at the sizes of the solutions located by the GA. For the trail problem M1 and M2 located FSMs which averaged between 11.32 (32K population, 30 seed, both *maxgens*, M1, figure 8.33) and 13.91 (16K population, 30 seed, *maxgen*=100, M2, figure 8.25) states. For all practical purposes with a 32-state genome, the genome used by the MTFs is $100 \cdot (11.32/32) \approx 35.4\%$ to $100 \cdot (13.91/32) \approx 43.5\%$ of the size of the genome used by the other methods since all the relevant data have been moved to the front of the genome. Recall that for the other methods the relevant data are spread across the genome. Thus with a 453-bit

genome the MTFs tend to produce significantly shorter schemata. Also recall (sections 2.2.4 and 4.2) that shorter useful schemata are more likely to survive crossover and increase their presence in subsequent generations. Therefore, it is not unexpected that MTF is more efficient for the trail problem.

The data for the languages are much different. The genome for L2 and L2c allows for a maximum of 8 states. For these two languages, experimentation showed the range of the average number of states in a solution for the MTFs is 5.82 (L2c, M1, all *maxgens* except 25) to 6.29 (L2, M1, *maxgen*=50). For the remaining 16 languages the genome allows for a maximum of 16 states. For these 16 languages the range of the average number of states in a solution for the MTFs is 11.04 (L6, M2, *maxgen*=25 and L9, M2, *maxgen*=1000) to 13.44 (L10c, M2, *maxgen*=25 and 75). Thus for the 18 languages, $100 \cdot (11.04/16) \approx 69\%$ to $100 \cdot (13.44/16) \approx 84\%$ of the genome's 59 or 148 bits is being used by MTF as opposed to 35.4% to 43.5% of 453 bits for the trail problem. The GA utilizing the MTFs for the languages do not get sufficiently smaller schemata than the other methods to cause them to consistently perform better than B1.

The next chapter summarizes the conclusions of this thesis and presents suggestions for further research.

>>>●<<<

Chapter 10

Conclusions

Jefferson et al. (1992) used a GA to locate an FSM which represents a strategy to successfully complete the John Muir trail (figures 4.1 and 4.3, section 4.3) with the application of a maximum of 200 pairs of transition-output rules. Traversing this trail from start to finish becomes progressively harder due to the increasing occurrence of unmarked sections along the trail. A successful trail following strategy was located by Jefferson et al.'s GA using a genome (figure 4.6 of section 4.2) which allowed for an FSM with a maximum of 32 states. Based on schema theory (sections 2.2.4 and 4.2), the nature of the FSM within the genome they used should inhibit the GA's progress towards a solution. In an attempt to enhance schema growth and provide a more efficient search with a GA, two FSM reorganization operators (SFS and MTF, sections 5.1 and 5.2 respectively) were designed for this research project. These operators were applied to successive generations of FSMs bred by a GA to see if one or both of these operators would hasten the search for a solution. In addition, GAs with these operators were applied to a second problem area, that of language recognition based on the Tomita sets (1982).

Two complementary explorations were carried out in the course of this thesis. First, two reorganization operators (SFS and MTF, sections 5.1 and 5.2 respectively)

were developed, and experiments were performed (sections 8.1.1, 8.2.1, and 9.1) to determine their relative efficiency in a GA. The second aspect of the research consisted of the development of criteria to analyze the results. The next section summarizes the evaluation criteria. Then section 10.2 recounts the conclusions with respect to MTF and SFS efficiency. Finally section 10.3 suggests directions for future research.

10.1 Summary of evaluation criteria

Numerous experiments were carried out. First two experiments (section 8.1.1) used a GA to locate an FSM which when applied to the John Muir trail (figures 4.1 and 4.3 of section 4.2) could complete the trail within a given number of time steps. The experiments used different population sizes. Each experiment consisted of 14 methods. For these two experiments, detailed data were collected for each generation (figures 5.8 and 5.9 of section 5.3).

The second set of experiments augmented the first two experiments with four more experiments. This was the result of using yet a third population size, and using both the original seed set and a completely different set of seeds for the three different population sizes. For this second set of experiments data were retained only for the outcome of each run (figure 7.1 of section 7.1).

The third and final set of experiments applied the GA to 18 languages (section 9.1.1). Each experiment consisted of eight methods (section 9.1.2). As for the second set of experiments for the trail problem, data were collected only for the

outcome of each run (figure 7.1 of section 7.1).

The data generated by the experiments were examined at several different levels starting with the level of comparing individual runs, and ending with a broad view across all of the experiments experiments for a given problem.

Initially, data were collected on a per generation basis for each run. Examples of this output appear in figures 5.8 and 5.9 of section 5.3. Three types of graphs were designed to compare individual runs from different methods. The individual runs for different methods were compared based on the same initial population, evolutionary progress, and location of an optimal individual. The complete description of these graphs appears in section 7.2. The graphs appear in appendix C, and the data evaluation is in section 8.1.2.

Moving towards a more general view, another type of graph was utilized; here, the highest fitness appearing in each generation was first averaged over all the runs for a method. Then these averages were graphed (description in section 7.2, graphs and data evaluation in section 8.1.2). Both the graphs comparing individual runs and the graphs comparing averages were used to analyze the evolutionary process (section 8.1.2).

While the evolutionary process gives some insight to the relative performance of the methods, the real concern is whether any method(s) will consistently find a solution to a problem first, based on number of generations and amount of processor time. For this analysis the focus moved from the per generation level to the per run level and per method level. An example of the output generated for this analysis

appears in figure 7.1 of section 7.1. Criteria to evaluate efficiency were designed around this type of data. G_{AV} is the average number of generations it took the successful runs to find a solution. $T_{AV:S}$ is the corresponding average processor time. Both of these measures apply only to the successful runs for a method: neither takes into consideration the fact that some runs failed to find a solution. Let F be the number of failed runs for a method. For each F a corresponding K (section 7.3.10.1) was calculated. K is the expected number of runs needed to find a solution with some selected probability. For this thesis that probability is .99.

Angeline and Pollack (1993) compared two different methods by ordering the successful runs within each method based on the number of generations used to find a solution, and comparing the i^{th} runs on the two lists. The same was done for this thesis, but only the best five runs based on the generation in which a solution was found were compared. Also for this thesis, the same was done using the processor time as the basis for ordering.

Comparing the average number of generations or average processor time needed by a successful run to find a solution combined with the same for the failed runs gave rise to two sets of formulas for G_{μ} (the mean number of generations needed to find a solution) and T_{μ} (the corresponding mean processor time). Formulas are developed for G_{μ} and T_{μ} for the sample (section 7.3.9). Another set of formulas for G_{μ} and T_{μ} (sections 7.3.5 and 7.3.7) uses the data from the sample to predict these means for the population.

For seven criteria (K , G_{AV} , $T_{AV:S}$, two G_{μ} s, and two T_{μ} s), speedup over the

benchmark was defined as the number of generations or amount of processor time for the benchmark divided by the same for the method being compared to the benchmark. For each of these criteria, the speedup was used to define efficiency.

Another criterion was designed which combined the number of failures which could precede a successful run (or the corresponding processor time), the number of generations used by a successful run to locate a solution (or the corresponding processor time), and a pairwise comparison of the values for two methods (sections 8.2.3.3 and 8.2.3.4). The probability that each of the two methods being compared would get a solution before the other was then calculated in addition to the percent increase of the probability of the better of the two methods over the poorer performer.

The methods within each experiment were ordered/ranked for each of these criteria. Those methods which consistently or almost consistently ranked higher than the benchmark across the nine criteria, combined with a high degree of confidence from a *U Test* (when applicable), were considered more efficient than the benchmark. In addition it was suspected that the MTF algorithm would locate solutions with fewer states. To verify this hypothesis the number of states in the solutions were recorded and averaged (figures 5.8 and 5.9 of section 5.3). The successful runs were sorted by increasing machine size and the i^{th} runs on the sorted lists for the different methods were compared. Ranking was induced by the lists for the i^{th} runs, and by average machine size.

These criteria worked well for the six experiments undertaken for the second

part of the research which evaluated the efficiency/convergence for the trail following problem. For the last set of 18 experiments (section 9.1) for the language recognition problem there was much more data than for the six experiments of the trail following problem, and a better way of getting a clear view of the relative performance of the different methods was needed.

To evaluate the massive amount of data produced by this last round of experiments, the speedups or percent increases were summarized for each of the nine criteria (F and K , G_{AV} , $T_{AV:S}$, two $G_{\mu s}$, two $T_{\mu s}$, and probability of getting a solution first for the number of generations and for the amount of processor time). In addition, for average machine size, the percent decrease with respect to $B1$ and U Test degree of confidence were also summarized. The worksheets containing these summaries appear in appendix D. They give a broad view of the performance of each method with respect to the benchmark. For each criterion, the methods which perform better than the benchmark are indicated. Those methods which generally perform better than the benchmark across the nine criteria are recommended for the experiment. Those methods which perform better than the benchmark across the 18 experiments are considered more efficient than the benchmark.

The evaluation procedures discussed in this section were applied to the data from the experiments. The conclusions drawn from the data generated by the experiments are presented in the next section.

10.2 Conclusions

The first two sets of experiments (sections 8.1.1 and 8.2.1) applied 14 methods (section 8.1.1) to the trail following problem (chapter 4). For the benchmark the work of Jefferson et al. (1992) was recreated. Then, either the SFS algorithm (section 5.1) or the MTF algorithm (section 5.2) was added to the benchmark. SFS uses a multi-part mathematical function to standardize the future or next states for each present state/input pair. MTF systematically moves the relevant data to the front of the genome. Each of these algorithms reorganized the data within the genome during the execution of the GA. The two reorganization operators were designed with the intention of enhancing schemata development, hence speeding up the evolutionary process and ultimately locating a solution faster (in fewer generation and/or with less processor time) than the benchmark.

There was a concern that the two new reorganization operators might tend to cause runs to converge prematurely (section 5.3). To counter this possible tendency, competition (section 5.3) was implemented. Each method (benchmark, SFS, MTF, and four variants of MTF) was executed both with and without competition giving rise to a total of 14 methods.

Using the data from the experiments with the trail problem, the SFS algorithm does not exhibit any consistent advantage over the benchmark. On the other hand MTF almost consistently performs better than the benchmark. These conclusions are based on an analysis of graphs (section 8.1.2) used to evaluate the evolutionary process, and rankings of the methods based on numerous criteria (sections 7.3.10.2,

and 7.3.10.3).

Experiments were then carried out on a set of 18 language recognition problems (section 9.1). For this set of experiments eight methods were tested. Across all of the experiments, the trail following experiments and the language recognition experiments, MTF consistently located FSM solutions with fewer states than the benchmark. Early results did not replicate the performance of MTF on the trail problem; sometimes MTF prevailed and sometimes SFS prevailed. Consequently a hybrid method (section 9.1.2) was developed. This hybrid applied MTF and SFS to alternating generations. The benchmark, SFS, MTF, and the hybrid were applied to the 18 languages, each without and then with competition.

Based on the data from the 18 language recognition experiments alone, no method consistently performed more efficiently than the benchmark. B1 was deemed more efficient than the other methods for two of the languages. The reason for this divergence from the results of the trail following problem (section 9.4) is based on schema theory (sections 2.2.4 and 4.2). On the average for the trail problem, the effective size of the genome used by MTF was 35.4% to 43.5% that of the genome used by the other methods (section 9.4). With the 453-bit genome (figure 4.6 of section 4.2) used for the trail problem, useful schemata are generally significantly shorter for MTF than for the other methods. Shorter useful schemata that lead to solutions are more likely to increase their representation in successive generations (sections 2.2.4 and 4.2). Consequently MTF was able to speed up the evolutionary process. Hence MTF performed more efficiently in the trail following experiments.

For the language recognition experiments the effective genome was 69% to 84% the size of the genome used by the other methods (section 9.4). With a 59-bit genome for L2 and L2c or 148-bit genome for the remaining languages (section 9.1.2), there is not a significant difference in the length of useful schemata produced by the different methods. Hence the performance advantage realized by MTF in the trail following problem did not carry over to the language recognition problem.

For the language recognition problem using the augmented Tomita sets, the parameters were selected (section 9.1.2) based on prior knowledge. In particular, the number of states, frequently the maximum number of states, needed for the solutions was known. Too large of a genome was found to produce solutions in generation 0. Maximum machine size was limited based on that knowledge. Maximum machine size was further limited to make the problems sufficiently difficult for the GA to permit the collection of useable data. (Recall that for two of the languages such a set of parameters was not found.) A researcher looking for a solution to a problem does not have this prior knowledge of expected machine size and would want to allow for a sufficiently large genome. For example, if the solution to a problem requires a 20-state FSM at minimum, a GA utilizing a genome allowing for a maximum of 16 states cannot locate a solution. Under these circumstances, the researcher would tend to allow for a much larger FSM than would be needed to ensure that the GA has a chance to locate a solution to the problem.

The difference between limiting machine size (as was done for the language recognition problem) and allowing for an excessive machine size (as would be

expected in applications) would tend to make a GA containing the MTF algorithm a more reasonable choice for an application. MTF also offers the advantage of finding smaller solutions. The decision as to whether to implement competition with MTF is not as clear. Competition requires an additional parameter (number of competitions, section 5.3). It is not known whether the value chosen for this parameter would greatly alter the performance of MTF with competition. Until the effect of the number of competitions on the efficiency of MTF with competition is determined, MTF without competition is probably a better choice for an application.

10.3 Future research

The experiments and conclusions presented in this thesis raise numerous questions. In this final section, follow-up research is suggested.

The experiments examined the sensitivity of the efficiency of the different methods to only two parameters: population size and the maximum number of generations that the GA was permitted to breed. Sensitivity to other parameters, especially to the number of competitions (when competition is implemented) should be examined. The point was made in the previous section that for an application of a GA to find a solution to a problem, a researcher probably will not select an optimal set of values for the parameters. Obviously methods which are more efficient than the benchmark across a wide range of parameters are more desirable.

More exploration is needed to determine the trade-off between improved efficiency of MTF when a large part of the genome is not used versus the additional

work that is done with a longer genome which allows for more states and which needs a larger population size. For this type of experimentation a researcher could perhaps consider a GA with MTF with a larger genome, and to compensate for a larger genome space this experiment should use a larger population size. It will be important to determine whether such a GA will examine **fewer individuals** (as opposed to fewer generations) and/or use less processor time than a benchmark without MTF, and with a smaller genome and smaller population size. The evaluation criteria involving the number of generations would have to be modified to give the total number of individuals processed. This can easily be done by multiplying the number of generations by the population size.

Two hybrids were examined. One (which was not previously referred to as a hybrid) applied MTF to generation 0 and then to every i^{th} generation after that, $i \in \{2,3,4,5\}$. This is actually a hybrid because it combines applications of the benchmark with applications of MTF. The other hybrid applied MTF and SFS to alternate generations. Hybrids which start off with one method to give the evolutionary process a jump-start, then finish with the benchmark (which does less work) should be explored.

The most detailed data collected and analyzed was on the per generation level. Examination of the progression of genomes from generation to generation, that is an examination at the level of the breeding process, would be interesting. Such an exploration could give a better understanding of the effect of the different methods on the evolutionary process at the most basic level of the GA, and would

probably raise additional questions.

A question suggesting another direction for experimentation is based on the fact that MTF locates smaller machines. If the fitness function for MTF were altered to favor smaller machines with the same fitness (as fitness was defined before the alteration), would the GA be more efficient than the GA with the unaltered fitness function? The results of the research for this thesis suggests that the answer to this question will depend on how much of a reduction in machine size can be realized using a fitness function favoring smaller machines.

The general conclusions of this thesis are that MTF locates smaller machines, and when the genome is much longer than the solutions found by the GA, the GA with MTF is more efficient than the one without MTF. It was also pointed out that a researcher looking for an FSM solution to a problem would tend to allow for an excessive number of states rather than chance selecting a number corresponding to a genome space which contains no solution. Under these conditions this thesis suggests that MTF tends to be more efficient than the other methods.

Based on the knowledge available as a result of this thesis project, a GA with MTF and without competition is recommended to locate an FSM solution to a problem with an FSM genome similar to those used in this thesis. The questions raised in this section indicate that the research presented in this thesis can lead to a more comprehensive understanding of GAs utilizing FSM genomes.

>>>●<<<

Appendix A: SFS walk-through

The reorganization of the FSM in table A.1 offers more insight to SFS. In the following discussion **to_be_processed2** and **queue2** are used interchangeably. Similarly **to_be_processed3** and **queue3** are used interchangeably.

A simple summary of **step2** clarifies the SFS reorganization. When a state is removed from **queue2** for processing, for that state and a given input get the next state number from the **next_state** table:

- 1) If the next state is **used**, it has already been reassigned. An attempt to reassign it at this point will undo the previous reassignment for that state. A previous reassignment is never changed by SFS. Therefore when the next state is **used** no further processing is needed for the present state-input pair being processed.
- 2) When the next state is not **used**, calculate the desired next state (lines 7-11, figure 5.5). If the desired next state is **used** there is a conflict of state assignment. An attempt to reassign it at this point will undo the previous reassignment for that state. Since **step2** cannot reassign the next state, the present state-input pair is put on **queue3** (lines 12-16, figure 5.5) for reassignment by step 3.
- 3) Only when both the next state and the desired next state are not **used** will they be interchanged (lines 17-22, figure 5.5).

It is suggested that the reader duplicate table A.1 and append a list of the states which do not appear on the table. Changing the states as indicated in the following description will make it easier to follow the walk-through. It is very difficult to follow the progress of the algorithm without a complete updated table available. Updated tables are periodically provided indicating only those lines of the table which have been changed with the changes on each line in boldface. Note that the program exchanges the data in the two affected rows of the **next_state** and **output** tables when present state names are exchanged, but in the walk-through

Appendix A continued

315

which follows the present state names are exchanged in table A.1.

Start state: q8

Line #	present state	next state/output for input=0	next state/output for input=1
1	q8	q19/3	q16/3
2	q0	q13/2	q21/2
3	q2	q21/3	q18/3
4	q3	q19/0	q11/1
5	q5	q11/0	q20/1
6	q6	q5/0	q15/1
7	q9	q2/3	q23/0
8	q10	q2/1	q0/2
9	q11	q25/1	q14/0
10	q13	q16/3	q8/0
11	q14	q9/1	q16/1
12	q15	q28/3	q20/2
13	q16	q0/0	q10/1
14	q18	q11/3	q9/1
15	q19	q13/0	q18/2
16	q20	q13/2	q3/0
17	q21	q0/2	q3/0
18	q23	q10/0	q6/2
19	q25	q30/0	q25/1
20	q26	q20/3	q15/0
21	q28	q18/1	q26/0
22	q30	q3/1	q16/2

Table A.1: FSM before SFS

When **SFS** is applied to table A.1, first step 1 interchanges q_0 and q_8 to make q_0 the start state (lines 6-8, figure 5.6). As a result of this exchange the next states for q_0 become q_{19} and q_{16} , and for q_8 they become q_{13} and q_{21} . Also this exchange changes the next state (lines 5-11, figure 5.2) of q_{10} with input 1 to q_8 , of q_{13} with input 1 to q_0 , of q_{16} with input 0 to q_8 , and of q_{21} with input 0 to q_8 . q_0 is tagged as **used** and put on queue2 for processing by **step2** (lines 7 and 9-10, figure 5.6). When this is completed (table A.2)

used: q_0 , and
to_be_processed2: q_0 .

Start state: q_0

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
1	q_0	$q_{19}/3$	$q_{16}/3$
2	q_8	$q_{13}/2$	$q_{21}/2$
8	q_{10}	$q_2/1$	$q_8/2$
10	q_{13}	$q_{16}/3$	$q_0/0$
13	q_{16}	$q_8/0$	$q_{10}/1$
17	q_{21}	$q_8/2$	$q_3/0$

Table A.2

step2 is then invoked (line 13, figure 5.6). q_0 is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_0 with input 0 is q_{19} . q_{19} is not **used** and should be $2 \times 0 + 0 + 1 \rightarrow q_1$ (line 7, figure 5.5). Since q_1 is not **used**, q_1 and q_{19} are interchanged (line 18, figure 5.5). One effect of this exchange

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
1	q0	q1/3	q2/3
3	q16	q21/3	q18/3
4	q3	q1/0	q11/1
7	q9	q16/3	q23/0
8	q10	q16/1	q8/2
10	q13	q2/3	q0/0
11	q14	q9/1	q2/1
13	q2	q8/0	q10/1
15	q1	q13/0	q18/2
22	q30	q3/1	q2/2
—	q19	—	—

Table A.3

is that the next states for q_1 become q_{13} and q_{18} . The next states for q_{19} are irrelevant because after the exchange is complete q_{19} is not reachable. In actuality the irrelevant states are retained in the table; they are just omitted from this discussion. The next state of q_0 with input 0 is changed to q_1 (lines 5-11, figure 5.2). The exchange of states q_1 and q_{19} also changes the next state (lines 5-11, figure 5.2) of q_3 with input 0 to q_1 . q_1 is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_0 with input 1, the next state for q_0 with input 1 is q_{16} . q_{16} is not **used** and should be $2 \times 0 + 1 + 1 \rightarrow q_2$ (line 7, figure 5.5). Since q_2 is not **used**, q_2 and q_{16} are interchanged (line 18, figure 5.5). This exchange causes the next states for q_2 to become q_8 and q_{10} , and for q_{16} they

become q_{21} and q_{18} . The next state of q_0 with input 1 is changed to q_2 (lines 5-11, figure 5.2). The exchange of states q_2 and q_{16} changes the next state (lines 5-11, figure 5.2) of q_9 with input 0 to q_{16} , of q_{10} with input 0 to q_{16} , of q_{13} with input 0 to q_2 , of q_{14} with input 1 to q_2 , and of q_{30} with input 1 to q_2 . q_2 is tagged as **used** and put on queue2 for processing by **step2** (lines 18-20, figure 34). When this is completed (table A.3)

used: q_0 - q_2 ; and
to_be_processed2: q_1 and q_2 .

Then q_1 is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_1 with input 0 is q_{13} . q_{13} is not **used** and should be $2 \times 1 + 0 + 1 \rightarrow q_3$ (line 7, figure 5.5). Since q_3 is not **used**, q_3 and q_{13} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_3 become q_2 and q_0 , and for q_{13} they become q_1 and q_{11} . The next state of q_1 with input 0 is changed to q_3 (lines 5-11, figure 5.2). The exchange of states q_3 and q_{13} changes the next state (lines 5-11, figure 5.2) of q_8 with input 0 to q_3 , of q_{21} with input 1 to q_{13} , and of q_{30} with input 0 to q_{13} . In addition, the next states (lines 5-11, figure 5.2) of q_{20} are changed to q_3 and q_{13} . q_3 is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_1 with input 1, the next state for q_1 with input 1 is q_{18} . q_{18} is not **used** and should be $2 \times 1 + 1 + 1 \rightarrow q_4$ (line 7, figure 5.5). Since q_4 is not **used**, q_4 and q_{18} are interchanged (line 18, figure 5.5). One outcome of this exchange is that the next states for q_4 become q_{11} and q_9 . The next states for q_{18} are irrelevant because after the exchange is complete q_{18} is not reachable. The next state of q_1 with input 1 is changed to q_4 (lines 5-11, figure 5.2).

Appendix A continued

319

Also the exchange of states q_4 and q_{18} changes the next state (lines 5-11, figure 5.2) of q_{16} with input 1 to q_4 , and of q_{28} with input 0 to q_4 . q_4 is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.4)

used: q_0 - q_4 : and
to_be_processed2: q_2 , q_3 , and q_4 .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
2	q_8	$q_3/2$	$q_{21}/2$
3	q_{16}	$q_{21}/3$	$q_4/3$
4	q_{13}	$q_1/0$	$q_{11}/1$
10	q_3	$q_2/3$	$q_0/0$
14	q_4	$q_{11}/3$	$q_9/1$
15	q_1	$q_3/0$	$q_4/2$
16	q_{20}	$q_3/2$	$q_{13}/0$
17	q_{21}	$q_8/2$	$q_{13}/0$
21	q_{28}	$q_4/1$	$q_{26}/0$
22	q_{30}	$q_{13}/1$	$q_2/2$
—	q_{18}	—	—

Table A.4

q_2 is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_2 with input 0 is q_8 . q_8 is not **used** and should be $2 \times 2 + 0 + 1 \rightarrow q_5$ (line 7, figure 5.5). Since q_5 is not **used**, q_5 and q_8 are interchanged (line 18, figure 5.5). As a side effect of this exchange the next states for q_5 become

q_3 and q_{21} , and for q_8 they become q_{11} and q_{20} . The next state of q_2 with input 0 is changed to q_5 (lines 5-11, figure 5.2). The exchange of states q_5 and q_8 also changes the next state (lines 5-11, figure 5.2) of q_6 with input 0 to q_8 , of q_{10} with input 1 to q_5 , and of q_{21} with input 0 to q_5 . q_5 is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). (See table A.5.) Moving on to q_2

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
2	q5	$q_3/2$	$q_{21}/2$
5	q8	$q_{11}/0$	$q_{20}/1$
6	q_6	q8/0	$q_{15}/1$
8	q_{10}	$q_{16}/1$	q5/2
13	q_2	q5/0	$q_{10}/1$
17	q_{21}	q5/0	$q_{13}/2$

Table A.5

with input 1, the next state for q_2 with input 1 is q_{10} . q_{10} is not **used** and should be $2 \times 2 + 1 + 1 \rightarrow q_6$ (line 7, figure 5.5). Since q_6 is not **used**, q_6 and q_{10} are interchanged (line 18, figure 5.5). This exchange causes the next states for q_6 to become q_{16} and q_5 , and for q_{10} they become q_8 and q_{15} . The next state of q_2 with input 1 is changed to q_6 (lines 5-11, figure 5.2). In addition, the next states (lines 5-11, figure 5.2) of q_{23} are changed to q_6 and q_{10} . q_6 is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.6)

used: q_0 - q_6 ; and

to_be_processed2: q_3 , q_4 , q_5 , and q_6 .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
6	q10	q8/0	q15/1
8	q6	q16/1	q5/2
13	q2	q5/0	q6/1
18	q23	q6/0	q10/2

Table A.6

Then q_3 is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_3 with input 0 is q_2 . Since q_2 is **used**, no further processing is needed for q_3 with input 0 (line 6, figure 5.5). Moving on to q_3 with input 1, the next state for q_3 with input 1 is q_0 . Since q_0 is **used**, no further processing is needed for q_3 with input 1 (line 6, figure 5.5). At this point (no changes to table)

used: q_0 - q_6 ; and
to_be_processed2: q_4 , q_5 , and q_6 .

q_4 is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_4 with input 0 is q_{11} . q_{11} is not **used** and should be $2 \times 4 + 0 + 1 \rightarrow q_9$ (line 7, figure 5.5). Since q_9 is not **used**, q_9 and q_{11} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_9 become q_{25} and q_{14} , and for q_{11} they become q_{16} and q_{23} . The next state of q_4 with input 0 is changed to q_9 (lines 5-11, figure 5.2). The exchange of states q_9 and q_{11} changes the next state (lines 5-11, figure 5.2) of q_4 with input 1 to q_{11} , of q_8 with input 0 to q_9 , of q_{13} with input 1 to q_9 , and of q_{14} with input 0 to q_{11} . q_9 is tagged as **used** and

put on queue2 for processing by **step2** (lines 19-20, figure 5.5). (See table A.7.)

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
4	q13	q1/0	q9/1
5	q8	q9/0	q20/1
7	q11	q16/3	q23/0
9	q9	q25/1	q14/0
11	q14	q11/1	q2/1
14	q4	q9/3	q11/1

Table A.7

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
6	q11	q8/0	q15/1
7	q10	q16/3	q23/0
11	q14	q10/1	q2/1
14	q4	q9/3	q10/1
18	q23	q6/0	q11/2

Table A.8

Moving on to q₄ with input 1, the next state for q₄ with input 1 is q₁₁. q₁₁ is not **used** and should be 2×4+1+1→q₁₀ (line 7, figure 5.5). Since q₁₀ is not **used**, q₁₀ and q₁₁ are interchanged (line 18, figure 5.5). As a side effect of this exchange the next states for q₁₀ become q₁₆ and q₂₃, and for q₁₁ they become q₈ and q₁₅. The next state of q₄ with input 1 is changed to q₁₀ (lines 5-11, figure 5.2). Also the

exchange of states q_{10} and q_{11} changes the next state (lines 5-11, figure 5.2) of q_{14} with input 0 to q_{10} , and of q_{23} with input 1 to q_{11} . q_{10} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.8)

used: q_0 - q_6 and q_9 - q_{10} ; and
to_be_processed2: q_5 , q_6 , q_9 , and q_{10} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
2	q_5	$q_3/2$	$q_{12}/2$
3	q_{16}	$q_{12}/3$	$q_4/3$
17	q_{12}	$q_5/2$	$q_{13}/0$
—	q_{21}	—	—

Table A.9

Then q_5 is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_5 with input 0 is q_3 . Since q_3 is **used**, no further processing is needed for q_5 with input 0 (line 6, figure 5.5). Moving on to q_5 with input 1, the next state for q_5 with input 1 is q_{21} . q_{21} is not **used** and should be $2 \times 5 + 1 + 1 \rightarrow q_{12}$ (line 7, figure 5.5). Since q_{12} is not **used**, q_{12} and q_{21} are interchanged (line 18, figure 5.5). One outcome of this exchange is that the next states for q_{12} become q_5 and q_{13} . The next states for q_{21} are irrelevant because after the exchange is complete q_{21} is not reachable. The next state of q_5 with input 1 is changed to q_{12} (lines 5-11, figure 5.2). The exchange of states q_{12} and q_{21} changes

the next state (lines 5-11, figure 5.2) of q_{16} with input 0 to q_{12} . q_{12} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.9)

used: q_0 - q_6 , q_9 - q_{10} , and q_{12} ; and
to_be_processed2: q_6 , q_9 , q_{10} , and q_{12} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
3	q_{13}	$q_{12}/3$	$q_4/3$
4	q_{16}	$q_1/0$	$q_9/1$
7	q_{10}	$q_{13}/3$	$q_{23}/0$
8	q_6	$q_{13}/1$	$q_5/2$
16	q_{20}	$q_3/2$	$q_{16}/0$
17	q_{12}	$q_5/2$	$q_{16}/0$
22	q_{30}	$q_{16}/1$	$q_2/2$

Table A.10

Then q_6 is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_6 with input 0 is q_{16} . q_{16} is not **used** and should be $2 \times 6 + 0 + 1 \rightarrow q_{13}$ (line 7, figure 5.5). Since q_{13} is not **used**, q_{13} and q_{16} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_{13} become q_{12} and q_4 , and for q_{16} they become q_1 and q_9 . The next state of q_6 with input 0 is changed to q_{13} (lines 5-11, figure 5.2). Also the exchange of states q_{13} and q_{16} changes the next state (lines 5-11, figure 5.2) of q_{10} with input 0 to q_{13} , of q_{12} with input 1 to q_{16} , of q_{20} with input 1 to q_{16} , and of q_{30} with input 0 to q_{16} . q_{13} is

tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_6 with input 1, the next state for q_6 with input 1 is q_5 . Since q_5 is **used**, no further processing is needed for q_6 with input 1 (line 6, figure 5.5). At this point (table A.10)

used: q_0 - q_6 , q_9 - q_{10} , and q_{12} - q_{13} ; and
to_be_processed2: q_9 , q_{10} , q_{12} , and q_{13} .

q_9 is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_9 with input 0 is q_{25} . q_{25} is not **used** and should be $2 \times 9 + 0 + 1 \rightarrow q_{19}$ (line 7, figure 5.5). Since q_{19} is not **used**, q_{19} and q_{25} are interchanged (line 18, figure 5.5). One outcome of this exchange is that the next states for q_{19} become q_{30} and q_{25} . The next states for q_{25} are irrelevant because after the exchange is complete q_{25} is not reachable. The next state of q_9 with input 0 is changed to q_{19} (lines 5-11, figure 5.2). The exchange of states q_{19} and q_{25} also changes the next state (lines 5-11, figure 5.2) of q_{19} with input 1 to q_{19} . q_{19} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_9 with input 1, the next state for q_9 with input 1 is q_{14} . q_{14} is not **used** and should be $2 \times 9 + 1 + 1 \rightarrow q_{20}$ (line 7, figure 5.5). Since q_{20} is not **used**, q_{20} and q_{14} are interchanged (line 18, figure 5.5). This exchange causes the next states for q_{14} to become q_3 and q_{16} , and for q_{20} they become q_{10} and q_2 . The next state of q_9 with input 1 is changed to q_{20} (lines 5-11, figure 5.2). The exchange of states q_{14} and q_{20} changes the next state (lines 5-11, figure 5.2) of q_8 with input 1 to q_{14} , of q_{15} with input 1 to q_{14} , and of q_{26} with input 0 to q_{14} . q_{20} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is

Appendix A continued

326

completed (table A.11)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , and q_{19} - q_{20} ; and
to_be_processed2: q_{10} , q_{12} , q_{13} , q_{19} , and q_{20} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
5	q_8	$q_9/1$	$q_{14}/1$
9	q_9	$q_{19}/1$	$q_{20}/0$
11	q_{20}	$q_{10}/1$	$q_2/1$
12	q_{15}	$q_{28}/3$	$q_{14}/2$
16	q_{14}	$q_3/2$	$q_{16}/0$
19	q_{19}	$q_{30}/0$	$q_{19}/1$
20	q_{26}	$q_{14}/3$	$q_{15}/0$
—	q_{25}	—	—

Table A.11

Then q_{10} is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{10} with input 0 is q_{13} . Since q_{13} is **used**, no further processing is needed for q_{10} with input 0 (line 6, figure 5.5). Moving on to q_{10} with input 1, the next state for q_{10} with input 1 is q_{23} . q_{23} is not **used** and should be $2 \times 10 + 1 + 1 \rightarrow q_{22}$ (line 7, figure 5.5). Since q_{22} is not **used**, q_{22} and q_{23} are interchanged (line 18, figure 5.5). A side effect of this exchange is that the next states for q_{22} become q_6 and q_{11} . The next states for q_{23} are irrelevant because after the exchange is complete q_{23} is not reachable. The next state of q_{10} with input 1 is changed to q_{22} (lines 5-11, figure 5.2). q_{22} is tagged as **used** and put on queue2 for

processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.12)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{19} - q_{20} , and q_{22} ; and
to_be_processed2: q_{12} , q_{13} , q_{19} , q_{20} , and q_{22} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
7	q_{10}	$q_{13}/3$	$q_{22}/0$
18	q_{22}	$q_6/0$	$q_{11}/2$
—	q_{23}	—	—

Table A.12

q_{12} is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{12} with input 0 is q_5 . Since q_5 is **used**, no further processing is needed for q_{12} with input 0 (line 6, figure 5.5). Moving on to q_{12} with input 1, the next state for q_{12} with input 1 is q_{16} . q_{16} is not **used** and should be $2 \times 12 + 1 + 1 \rightarrow q_{26}$ (line 7, figure 5.5). Since q_{26} is not **used**, q_{26} and q_{16} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_{16} become q_{14} and q_{15} , and for q_{26} they become q_1 and q_9 . The next state of q_{12} with input 0 is changed to q_{26} (lines 5-11, figure 5.2). Also the exchange of states q_{16} and q_{26} changes the next state (lines 5-11, figure 5.2) of q_{14} with input 1 to q_{26} , of q_{28} with input 1 to q_{16} , and of q_{30} with input 0 to q_{26} . q_{26} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.13)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{19} - q_{20} , q_{22} , and q_{26} ; and
to_be_processed2: q_{13} , q_{19} , q_{20} , q_{22} , and q_{26} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
4	q26	$q_1/0$	$q_9/1$
16	q_{14}	$q_3/2$	q26/0
17	q_{12}	$q_5/2$	q26/0
20	q16	$q_{14}/3$	$q_{15}/0$
21	q_{28}	$q_4/1$	q16/0
22	q_{30}	q26/1	$q_2/2$

Table A.13

Then q_{13} is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{13} with input 0 is q_{12} . Since q_{12} is **used**, no further processing is needed for q_{13} with input 0 (line 6, figure 5.5). Moving on to q_{13} with input 1, the next state for q_{13} with input 1 is q_4 . Since q_4 is **used**, no further processing is needed for q_{13} with input 1 (line 6, figure 5.5). At this point (no change to the table)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{19} - q_{20} , q_{22} , and q_{26} ; and
to_be_processed2: q_{19} , q_{20} , q_{22} , and q_{26} .

q_{19} is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{19} with input 0 is q_{30} . q_{30} is not **used** and should be $2 \times (32 - 2 - 19) + 0 + 1 \rightarrow q_{23}$ (line 11, figure 5.5). Note that a different part of the function is being used to determine the next state. Since q_{23} is not **used**, q_{23} and

q_{30} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_{23} become q_{26} and q_2 . The next states for q_{30} are irrelevant because after the exchange is complete q_{30} is not reachable. The next state of q_{19} with input 0 is changed to q_{23} (lines 5-11, figure 5.2). q_{23} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_{19} with input 1, the next state for q_{19} with input 1 is q_{19} . Since the next state is **used**, no further processing is needed for q_{19} with input 1 (line 6, figure 5.5). At this point (table A.14)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{19} - q_{20} , q_{22} - q_{23} , and q_{26} ; and
to_be_processed2: q_{20} , q_{22} , q_{26} , and q_{23} .

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
19	q_{19}	$q_{23}/0$	$q_{19}/1$
22	q_{23}	$q_{26}/1$	$q_2/1$
—	q_{30}	—	—

Table A.14

Then q_{20} is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{20} with input 0 is q_{10} . Since q_{10} is **used**, no further processing is needed for q_{20} with input 0 (line 6, figure 5.5). Moving on to q_{20} with input 1, the next state for q_{20} with input 1 is q_2 . Since q_2 is **used**, no further processing is needed for q_{20} with input 1 (line 6, figure 5.5). At this point (no changes to the table)

used: q₀-q₆, q₉-q₁₀, q₁₂-q₁₃, q₁₉-q₂₀, q₂₂-q₂₃, and q₂₆; and
to_be_processed2: q₂₂, q₂₆, and q₂₃.

q₂₂ is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q₂₂ with input 0 is q₆. Since q₆ is **used**, no further processing is needed for q₂₂ with input 0 (line 6, figure 5.5). Moving on to q₂₂ with input 1, the next state for q₂₂ with input 1 is q₁₁. q₁₁ is not **used** and should be $2 \times (32 - 2 - 22) + 1 + 1 \rightarrow q_{18}$ (line 11, figure 5.5). Since q₁₈ is not **used**, q₁₈ and q₁₁ are interchanged (line 18, figure 5.5). This exchange causes the next states for q₁₈ to become q₈ and q₁₅. The next states for q₁₁ are irrelevant because after the exchange is complete q₁₁ is not reachable. The next state of q₂₂ with input 1 is changed to q₁₈ (line 5-11, figure 5.2). q₁₈ is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.15)

used: q₀-q₆, q₉-q₁₀, q₁₂-q₁₃, q₁₈-q₂₀, q₂₂-q₂₃, and q₂₆; and
to_be_processed2: q₂₆, q₂₃, and q₁₈.

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
6	q18	q8/0	q15/1
18	q22	q6/0	q18/2
—	q11	—	—

Table A.15

Then q₂₆ is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q₂₆ with input 0 is q₁. Since q₁ is **used**, no further

processing is needed for q_{26} with input 0 (line 6, figure 5.5). Moving on to q_{26} with input 1, the next state for q_{26} with input 1 is q_9 . Since q_9 is **used**, no further processing is needed for q_{26} with input 1 (line 6, figure 5.5). At this point (no change to the table)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{18} - q_{20} , q_{22} - q_{23} , and q_{26} ; and
to_be_processed2: q_{23} and q_{18} .

q_{23} is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{23} with input 0 is q_{26} . Since q_{26} is **used**, no further processing is needed for q_{23} with input 0 (line 6, figure 5.5). Moving on to q_{23} with input 1, the next state for q_{23} with input 1 is q_2 . Since q_2 is **used**, no further processing is needed for q_{23} with input 1 (line 6, figure 5.5). At this point (no change to the table)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{18} - q_{20} , q_{22} - q_{23} , and q_{26} ; and
to_be_processed2: q_{18} .

Then q_{18} is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{18} with input 0 is q_8 . q_8 is not **used** and should be $2 \times (32 - 2 - 18) + 0 + 1 \rightarrow q_{25}$ (line 11, figure 5.5). Since q_{25} is not **used**, q_{25} and q_8 are interchanged (line 18, figure 5.5). One outcome of this exchange is that the next states for q_{25} become q_9 and q_{14} . The next states for q_8 are irrelevant because after the exchange is complete q_8 is not reachable. The next state of q_{18} with input 0 is changed to q_{25} (lines 5-11, figure 5.2). q_{25} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). Moving on to q_{18} with input 1, the next state for q_{18} with input 1 is q_{15} . q_{15} is not **used** and should be

$2 \times 0(32 - 2 - 18) + 1 + 1 \rightarrow q_{26}$ (line 11, figure 5.5). Since q_{26} is **used**, it cannot be assigned as the next state for q_{18} with an input of 1. Therefore $(q_{18}, 1)$ is put on queue3 for processing by step 3 (lines 12-16, figure 5.5). At this point (table A.16)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{18} - q_{20} , q_{22} - q_{23} , and q_{25} - q_{26} ;
to_be_processed2: q_{25} ; and
to_be_processed3: $(q_{18}, 1)$.

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
5	q₂₅	$q_9/0$	$q_{14}/1$
6	q_{18}	q₂₅/0	$q_{15}/1$
—	q₈	—	—

Table A.16

q_{25} is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q_{25} with input 0 is q_9 . Since q_9 is **used**, no further processing is needed for q_{25} with input 0 (line 6, figure 5.5). Moving on to q_{25} with input 1, the next state for q_{25} with input 1 is q_{14} . q_{14} is not **used** and should be $2 \times (32 - 2 - 25) + 1 + 1 \rightarrow q_{12}$ (line 11, figure 5.5). Since q_{12} is **used**, it cannot be assigned as the next state for q_{25} with an input of 1. Therefore $(q_{25}, 1)$ is put on queue3 for processing by step 3 (lines 12-16, figure 5.5). At this point (no change to the table)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{18} - q_{20} , q_{22} - q_{23} , and q_{25} - q_{26} ;
to_be_processed2: empty; and
to_be_processed3: $(q_{18}, 1)$ and $(q_{25}, 1)$.

Since queue2 is empty, control is returned to **SFS** (lines 13-15, figure 5.6), and step 3 is executed. Remove q_{18} with input 1 from queue3 (lines 16-18, figure 5.6) for processing by step 3. The next state for q_{18} with input 1 is q_{15} which is still not **used**. Looking at the used states (in the previous paragraph), the state, which is closest to q_{18} and which is not **used**, is q_{17} (lines 20-27, figure 5.6). Therefore q_{17} will become the **new_next_state** for q_{18} with input 1 (lines 24-27, figure 5.6). q_{17} has been selected because it is not **used**, therefore q_{17} and q_{15} can be interchanged (line 28, figure 5.6). As a side effect of this exchange the next states for q_{17} become q_{28} and q_{14} . The next states for q_{15} are irrelevant because after the exchange is complete q_{15} is not reachable. The next state of q_{18} with input 1 is changed to q_{17} (lines 5-11, figure 5.2). Also the exchange of states q_{15} and q_{17} changes the next state (lines 5-11, figure 5.2) of q_{16} with input 1 to q_{17} . q_{17} is tagged as **used** and put on queue2 for processing by **step2** (lines 29-31, figure 5.6). (Whenever possible, next states are assigned by **step2**.) When this is complete (table A.17)

used: q_0 - q_6 , q_9 - q_{10} , q_{12} - q_{13} , q_{17} - q_{20} , q_{22} - q_{23} , and q_{25} - q_{26} ;
to_be_processed2: q_{17} ; and
to_be_processed3: (q_{25} ,1).

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
6	q_{18}	$q_{25}/0$	$q_{17}/1$
12	q_{17}	$q_{28}/3$	$q_{14}/2$
20	q_{16}	$q_{14}/3$	$q_{17}/0$
—	q_{15}	—	—

Table A.17

step2 takes precedence over step 3 and queue2 is not empty. therefore **step2** is called (lines 32-33, figure 5.6). q_{17} is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5. The next state for q_{17} with input 0 is q_{28} . q_{28} is not **used** and should be $2 \times (32 - 2 - 17) + 0 + 1 \rightarrow q_{27}$ (line 11, figure 5.5). Since q_{27} is not **used**, q_{27} and q_{28} are interchanged (line 18, figure 5.5). As a result of this exchange the next states for q_{27} become q_4 and q_{16} . The next states for q_{28} are irrelevant because after the exchange is complete q_{28} is not reachable. The next state of q_{17} with input 0 is changed to q_{27} (lines 5-11, figure 5.2). q_{27} is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). (See table A.18.) Moving on to q_{17} with input 1, the next state for q_{17} with input 1 is q_{14} . q_{14} is not

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
12	q_{17}	$q_{27}/3$	$q_{14}/2$
21	q_{27}	$q_4/1$	$q_{16}/2$
—	q_{28}	—	—

Table A.18

used and should be $2 \times (32 - 2 - 17) + 1 + 1 \rightarrow q_{28}$ (line 11, figure 5.5). Since q_{28} is not **used**, q_{28} and q_{14} are interchanged (line 18, figure 5.5). This exchange causes the next states for q_{28} to become q_3 and q_{26} . The next states for q_{14} are irrelevant because after the exchange is complete q_{14} is not reachable. The next state of q_{17} with input 1 is changed to q_{28} (line 5-11, figure 5.2). The exchange of states q_{14} and

q₂₈ also changes the next state (lines 5-11, figure 5.2) of q₁₆ with input 0 to q₂₈, and of q₂₅ with input 1 to q₂₈. q₂₈ is tagged as **used** and put on queue2 for processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.19)

used: q₀-q₆, q₉-q₁₀, q₁₂-q₁₃, q₁₇-q₂₀, q₂₂-q₂₃, and q₂₅-q₂₈;
to_be_processed2: q₂₇ and q₂₈; and
to_be_processed3: (q₂₅.1).

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
5	q ₂₅	q ₉ /0	q ₂₈ /1
12	q ₁₇	q ₂₇ /3	q ₂₈ /2
16	q ₂₈	q ₃ /2	q ₂₆ /0
20	q ₁₆	q ₂₈ /3	q ₁₇ /0
—	q ₁₄	—	—

Table A.19

Then q₂₇ is removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q₂₇ with input 0 is q₄. Since q₄ is **used**, no further processing is needed for q₂₇ with input 0 (line 6, figure 5.5). Moving on to q₂₇ with input 1, the next state for q₂₇ with input 1 is q₁₆. q₁₆ is not **used** and should be $2 \times (32 - 2 - 27) + 1 + 1 \rightarrow q_8$ (line 11, figure 5.5). Since q₈ is not **used**, q₈ and q₁₆ are interchanged (line 18, figure 5.5). One outcome of this exchange is that the next states for q₈ become q₂₈ and q₁₇. The next states for q₁₆ are irrelevant because after the exchange is complete q₁₆ is not reachable. The next state of q₂₇ with input 1 is changed to q₈ (lines 5-11, figure 5.2). q₈ is tagged as **used** and put on queue2 for

processing by **step2** (lines 19-21, figure 5.5). When this is completed (table A.20)

used: q₀-q₆, q₈-q₁₀, q₁₂-q₁₃, q₁₇-q₂₀, q₂₂-q₂₃, and q₂₅-q₂₈;
to_be_processed2: q₂₈ and q₈; and
to_be_processed3: (q₂₅.1).

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
20	q ₈	q ₂₈ /3	q ₁₇ /0
21	q ₂₇	q ₄ /1	q ₈ /0
—	q ₁₆	—	—

Table A.20

q₂₈ is then removed from queue2 for processing by **step2** (lines 3-4, figure 5.5). The next state for q₂₈ with input 0 is q₃. Since q₃ is **used**, no further processing is needed for q₂₈ with input 0 (line 6, figure 5.5). Moving on to q₂₈ with input 1, the next state for q₂₈ with input 1 is q₂₆. Since q₂₆ is **used**, no further processing is needed for q₂₈ with input 1 (line 6, figure 5.5). At this point (no changes to the table)

used: q₀-q₆, q₈-q₁₀, q₁₂-q₁₃, q₁₇-q₂₀, q₂₂-q₂₃, and q₂₅-q₂₈;
to_be_processed2: q₈; and
to_be_processed3: (q₂₅.1).

Then q₈ is removed from queue2 for processing by **step2** (line 3-4, figure 5.5). The next state for q₈ with input 0 is q₂₈. Since q₂₈ is **used**, no further processing is needed for q₈ with input 0 (line 6, figure 5.5). Moving on to q₈ with input 1, the next state for q₈ with input 1 is q₁₇. Since q₁₇ is **used**, no further

processing is needed for q_8 with input 1 (line 6, figure 5.5). At this point (no changes to the table)

used: q_0 - q_6 , q_8 - q_{10} , q_{12} - q_{13} , q_{17} - q_{20} , q_{22} - q_{23} , and q_{25} - q_{28} ;
to_be_processed2: empty; and
to_be_processed3: (q_{25} ,1).

Since queue2 is empty, control is returned to **SFS** (figure 5.6), and step 3 is continued since the last call to **step2** was from within step 3 (lines 32-33, figure 5.6). Remove q_{25} with input 1 from queue3 (lines 16-18, figure 5.6) for processing by step 3. The next state for q_{25} with input 1 is q_{28} which is now **used**. Therefore no further processing is needed for q_{25} with input 1 (line 19, figure 5.6). The two queues are empty, hence the SFS reorganization of table A.1 is complete. Table A.22 contains the FSM of table A.1 after the SFS reorganization. Table A.21 correlates the states between tables A.1 and A.22.

State # from table A.1	0	2	3	5	6	8	9	10	11	13	14
State # from table A.22	5	13	26	25	18	0	10	6	9	3	20

State # from table A.1	15	16	18	19	20	21	23	25	26	28	30
State # from table A.22	17	2	4	1	28	12	22	19	8	27	23

Table A.21: Correlation of States between the FSMs in Tables A.1 and A.22

Start state: q0

Line # from table A.1	present state	next state/output for input=0	next state/output for input=1
1	q0	q1/3	q2/3
15	q1	q3/0	q4/2
13	q2	q5/0	q6/1
10	q3	q2/3	q0/0
14	q4	q9/3	q10/1
2	q5	q3/2	q12/2
8	q6	q13/1	q5/2
20	q8	q28/3	q17/0
9	q9	q19/1	q20/0
7	q10	q13/3	q22/0
17	q12	q5/2	q26/0
3	q13	q12/3	q4/3
12	q17	q27/3	q28/2
6	q18	q25/0	q17/1
19	q19	q23/0	q19/1
11	q20	q10/1	q2/1
18	q22	q6/0	q18/2
22	q23	q26/1	q2/2
5	q25	q9/0	q28/1
4	q26	q1/0	q9/1
21	q27	q4/1	q8/0
16	q28	q3/2	q26/0

Table A.22: FSM of table A.1 after SFS

Appendix B: Proofs for sums of infinite series

This appendix contains the proofs for equations 7.1 through 7.3 which appear in section 7.3.3.

The first of the three equations is for the sum of $\sum_{i=0}^{\infty} r^i$. This is the sum of an infinite geometric progression with a first term of 1 and ratio of r . Even though the formula for this sum is well known, its proof is included for completeness.

Equation 7.1: $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}, 0 < r < 1.$

Proof: Letting S be the required infinite sum, $S = 1 + r + r^2 + r^3 + \dots$. Multiplying both sides of this equation by r yields $rS = r + r^2 + r^3 + \dots$. Subtracting the equation with rS on the left, from the equation with only S on the left results in $(1-r)S = 1$. Solving for S completes the proof. $\square$

The two additional formulas which are needed are

Equation 7.2: $\sum_{i=1}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}, 0 < r < 1.$ and

Equation 7.3: $\sum_{i=1}^{\infty} i^2 r^{i-1} = \frac{r+1}{(1-r)^3}, 0 < r < 1.$

Two sets of proofs for these equations follow. The first analyzes infinite sequences; the second contains the classic proofs from calculus.

Proofs using sums of infinite sequences

In this section, the proofs for equations 7.2 and 7.3 use a series of formulas for sums of infinite sequences. The proofs systematically reduce sums of increasing complexity to sums for which closed forms were previously found.

Appendix B continued

Equation 7.2: $\sum_{i=1}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}, 0 < r < 1.$

Proof:

$$\begin{aligned}
 \sum_{i=1}^{\infty} i r^{i-1} &= 1 + 2r + 3r^2 + 4r^3 + 5r^4 + \dots \\
 &= 1 + r + r^2 + r^3 + r^4 + \dots \\
 &\quad + r + r^2 + r^3 + r^4 + \dots \\
 &\quad + r^2 + r^3 + r^4 + \dots \\
 &\quad + r^3 + r^4 + \dots \\
 &\quad \vdots \\
 &= \sum_{i=0}^{\infty} r^i + r \sum_{i=0}^{\infty} r^i + r^2 \sum_{i=0}^{\infty} r^i + r^3 \sum_{i=0}^{\infty} r^i + \dots \\
 &= (1 + r + r^2 + r^3 + r^4 + \dots) \sum_{i=0}^{\infty} r^i = \left(\sum_{i=0}^{\infty} r^i \right)^2.
 \end{aligned}$$

Applying equation 7.1 to this last sum completes the proof. $\square$

To prove equation 7.3, one lemma is needed in addition to equations 7.1 and 7.2. This lemma is derived as the last in a series of four lemmas.

Lemma B.1: $\sum_{i=k}^{\infty} k r^{i-1} = \frac{k r^{k-1}}{1-r}, k > 0, 0 < r < 1.$

Proof: $\sum_{i=k}^{\infty} k r^{i-1} = k r^{k-1} \sum_{i=k}^{\infty} r^{i-k}.$ Changing the value of the index of summation, and making a corresponding change in the summation term yields $k r^{k-1} \sum_{i=0}^{\infty} r^i.$ Applying equation 7.1 completes the proof. $\square$

When $k=1$, lemma B.1 reduces to equation 7.1. An alternate proof for this lemma can parallel the proof for equation 7.1.

Appendix B continued

Lemma B.2: $\sum_{i=k+1}^{\infty} (i-k)r^{i-1} = \frac{r^k}{(1-r)^2}, k \geq 0, 0 < r < 1.$

Proof: $\sum_{i=k+1}^{\infty} (i-k)r^{i-1} = r^k \sum_{i=k+1}^{\infty} (i-k)r^{i-k-1}.$ Changing the value of the index of summation, and correspondingly altering the summation term yields $r^k \sum_{i=1}^{\infty} ir^{i-1}.$

Applying equation 7.2 completes the proof. $\square$

Lemma B.3: $\sum_{i=k}^{\infty} ir^{i-1} = \frac{kr^{k-1}}{1-r} + \frac{r^k}{(1-r)^2}, k > 0, 0 < r < 1.$

Proof: $\sum_{i=k}^{\infty} ir^{i-1} = \sum_{i=k}^{\infty} (k+i-k)r^{i-1} = \sum_{i=k}^{\infty} kr^{i-1} + \sum_{i=k}^{\infty} (i-k)r^{i-1}.$

Since the first term of $\sum_{i=k}^{\infty} (i-k)r^{i-1}$ is 0, the index i can start at $k+1$. After incorporating this change, lemmas B.1 and B.2 are applied to complete the proof. $\square$

Lemma B.4:

$$\sum_{i=k}^{\infty} [i^2 - (k-1)i]r^{i-1} = \frac{kr^{k-1}}{1-r} + \frac{r^k}{(1-r)^2} + \sum_{i=k+1}^{\infty} (i^2 - ki)r^{i-1}, k > 0, 0 < r < 1.$$

Proof: $\sum_{i=k}^{\infty} [i^2 - (k-1)i]r^{i-1} = \sum_{i=k}^{\infty} (i + i^2 - ki)r^{i-1} = \sum_{i=k}^{\infty} ir^{i-1} + \sum_{i=k}^{\infty} (i^2 - ki)r^{i-1}.$

Apply lemma B.3 to $\sum_{i=k}^{\infty} ir^{i-1}.$ In $\sum_{i=k}^{\infty} (i^2 - ki)r^{i-1}$ the first term is 0; therefore the summation can start at $i=k+1$. These two changes complete the proof. $\square$

Equation 7.3: $\sum_{i=1}^{\infty} i^2 r^{i-1} = \frac{r+1}{(1-r)^3}, 0 < r < 1.$

Proof: Applying lemma B.4 with $k=1$ to $\sum_{i=1}^{\infty} i^2 r^{i-1}$ yields

$$\frac{1}{1-r} + \frac{r}{(1-r)^2} + \sum_{i=2}^{\infty} (i^2 - i)r^{i-1}.$$

Applying lemma B.4 with $k=2$ to this last summation yields

$$\frac{1}{1-r} + \frac{r}{(1-r)^2} + \frac{2r}{1-r} + \frac{r^2}{(1-r)^2} + \sum_{i=3}^{\infty} (i^2 - 2i)r^{i-1}.$$

Appendix B continued

Applying lemma B.4 again, but now with $k=3$ yields

$$\frac{1}{1-r} + \frac{r}{(1-r)^2} + \frac{2r}{1-r} + \frac{r^2}{(1-r)^2} + \frac{3r^2}{1-r} + \frac{r^3}{(1-r)^2} + \sum_{i=4}^{\infty} (i^2 - 3i) r^{i-1}.$$

Combining terms and applying lemma B.4 with $k=4$ yields

$$\frac{1+2r+3r^2}{1-r} + \frac{r+r^2+r^3}{(1-r)^2} + \frac{4r^3}{1-r} + \frac{r^4}{(1-r)^2} + \sum_{i=5}^{\infty} (i^2 - 4i) r^{i-1}.$$

Repeatedly applying lemma B.4 with successively increasing values of k and combining terms yields

$$\frac{1+2r+3r^2+4r^3+\dots}{1-r} + \frac{r}{(1-r)^2}(1+r+r^2+\dots) = \frac{\sum_{i=1}^{\infty} i r^{i-1}}{1-r} + \frac{r \sum_{i=0}^{\infty} r^i}{(1-r)^2}.$$

Applying equations 7.2 and 7.1 respectively, and combining terms completes the proof. $\square$

In the next section equations 7.2 and 7.3 are derived using calculus.

Classic proofs using calculus

Equations 7.2 and 7.3 are proven by sequentially taking derivatives and substituting previously obtained results.

Start with equation 7.1, $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$, $0 < r < 1$.

Taking the derivative of equation 7.1 with respect to r yields $\sum_{i=0}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}$.

Since $i r^{i-1} = 0$ when $i=0$, i can start at 1. Making this substitution yields equation

7.2 which is $\sum_{i=1}^{\infty} i r^{i-1} = \frac{1}{(1-r)^2}$, $0 < r < 1$.

Taking the derivative of equation 7.2 with respect to r yields

$$\sum_{i=1}^{\infty} i(i-1) r^{i-2} = \frac{2}{(1-r)^3}.$$

Appendix B continued

Multiplying both sides by r yields $\sum_{i=1}^{\infty} i(i-1)r^{i-1} = \sum_{i=1}^{\infty} i^2 r^{i-1} - \sum_{i=1}^{\infty} i r^{i-1} = \frac{2r}{(1-r)^3}$. Solving for $\sum_{i=1}^{\infty} i^2 r^{i-1}$ and applying equation 7.2 yields $\sum_{i=1}^{\infty} i^2 r^{i-1} = \frac{2r}{(1-r)^3} + \frac{1}{(1-r)^2} = \frac{r+1}{(1-r)^3}$ thus completing the proofs. $\square$

Appendix C

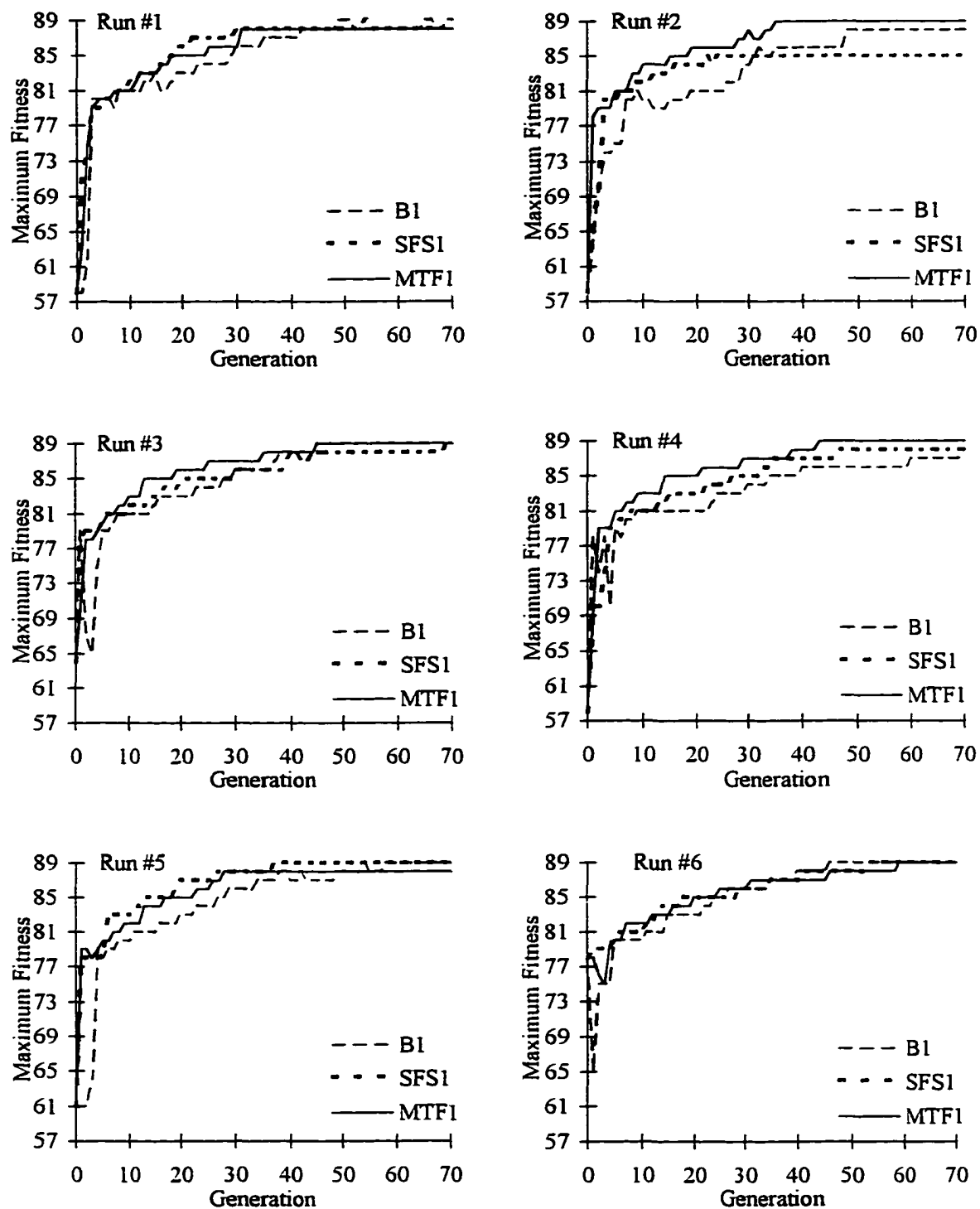


Figure C.1a: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

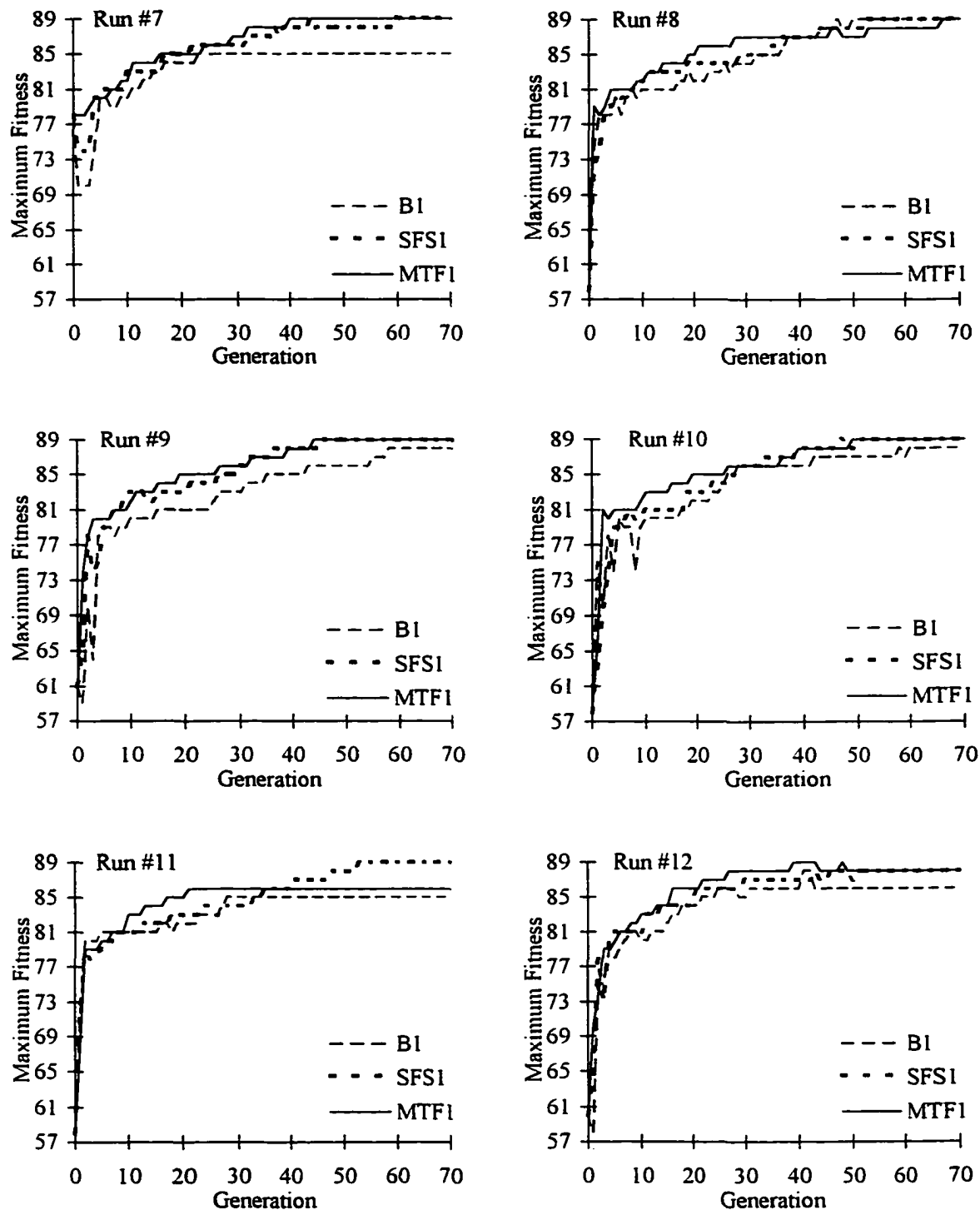


Figure C.1a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

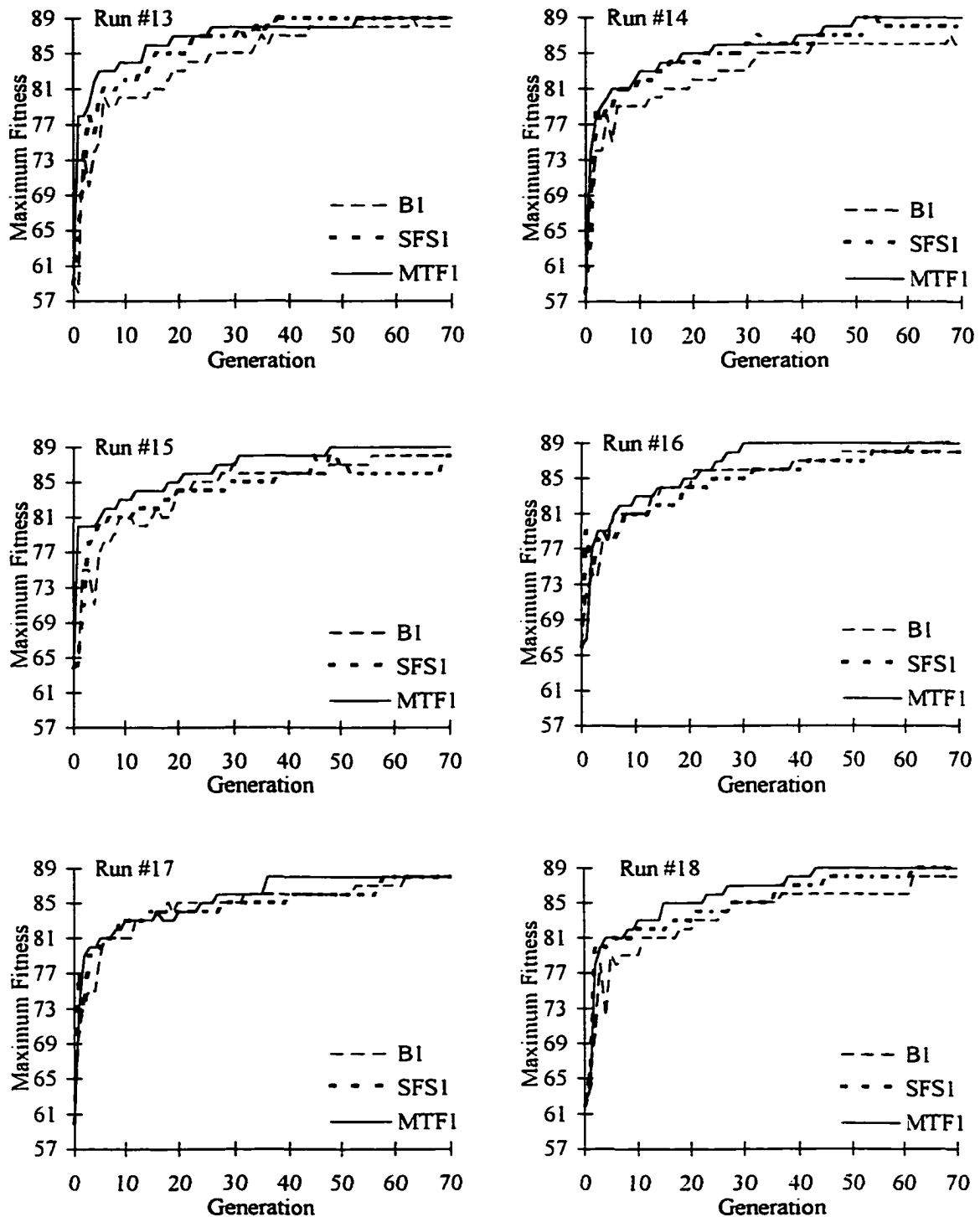


Figure C.1a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

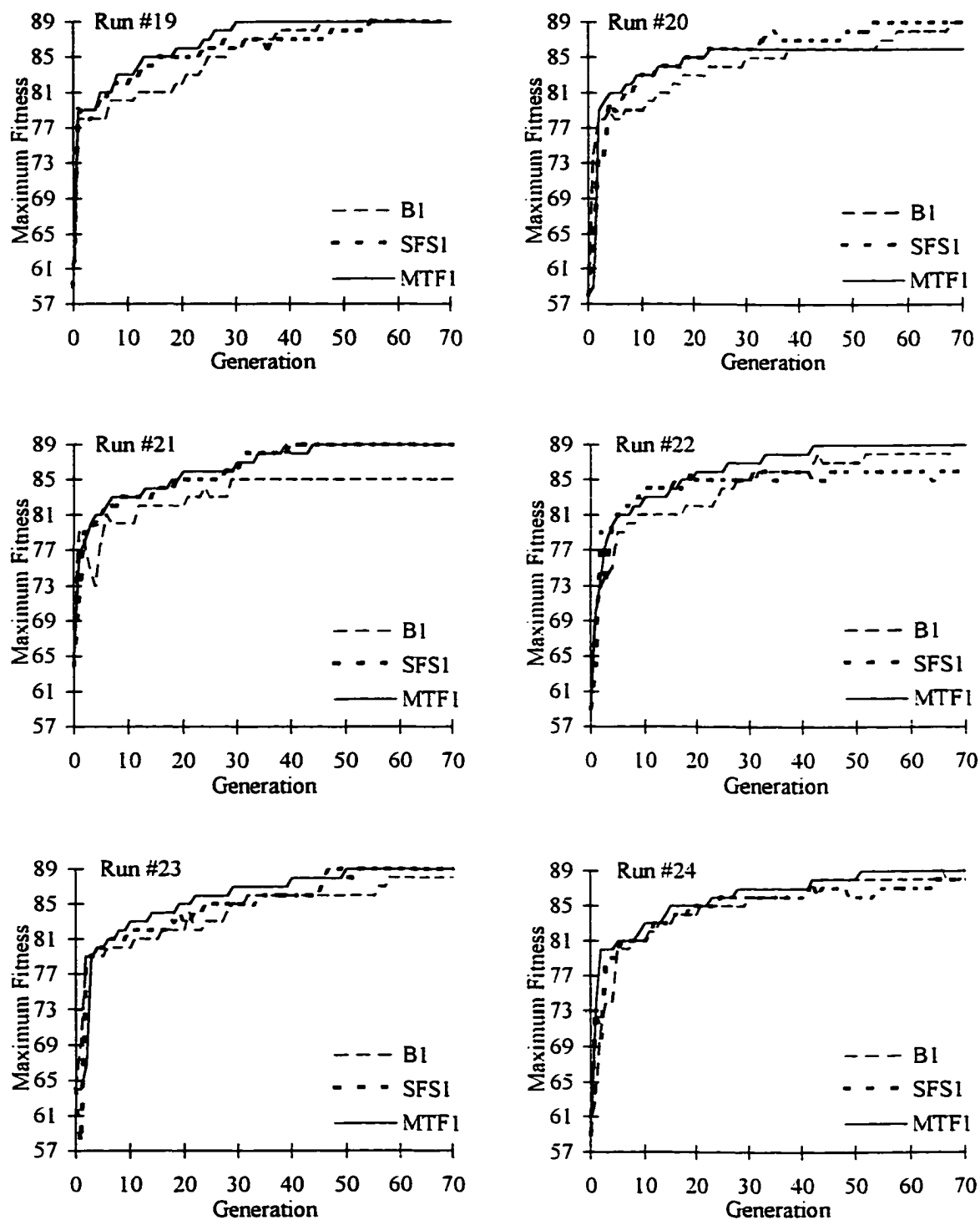


Figure C.1a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

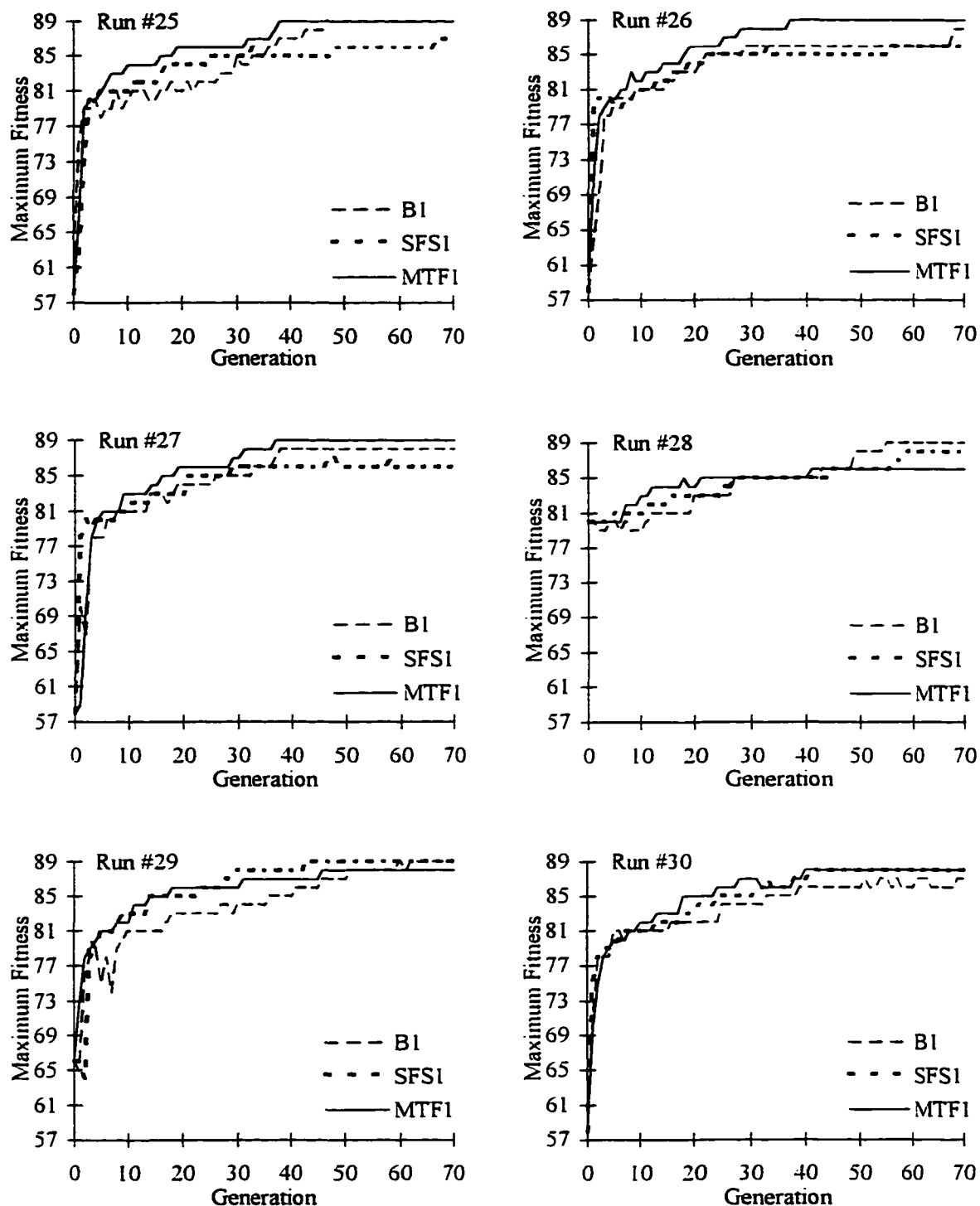


Figure C.1a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

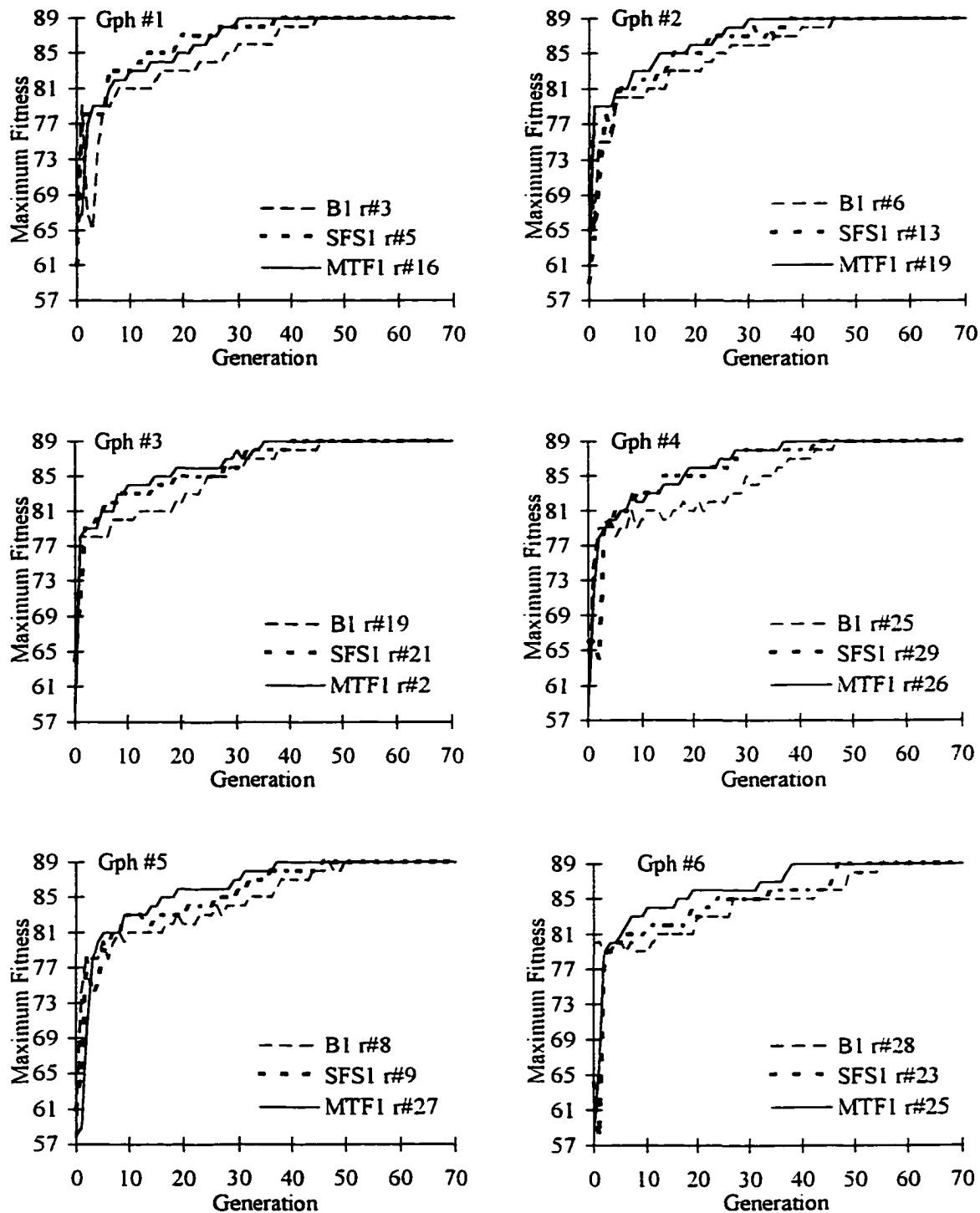


Figure C.1b: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

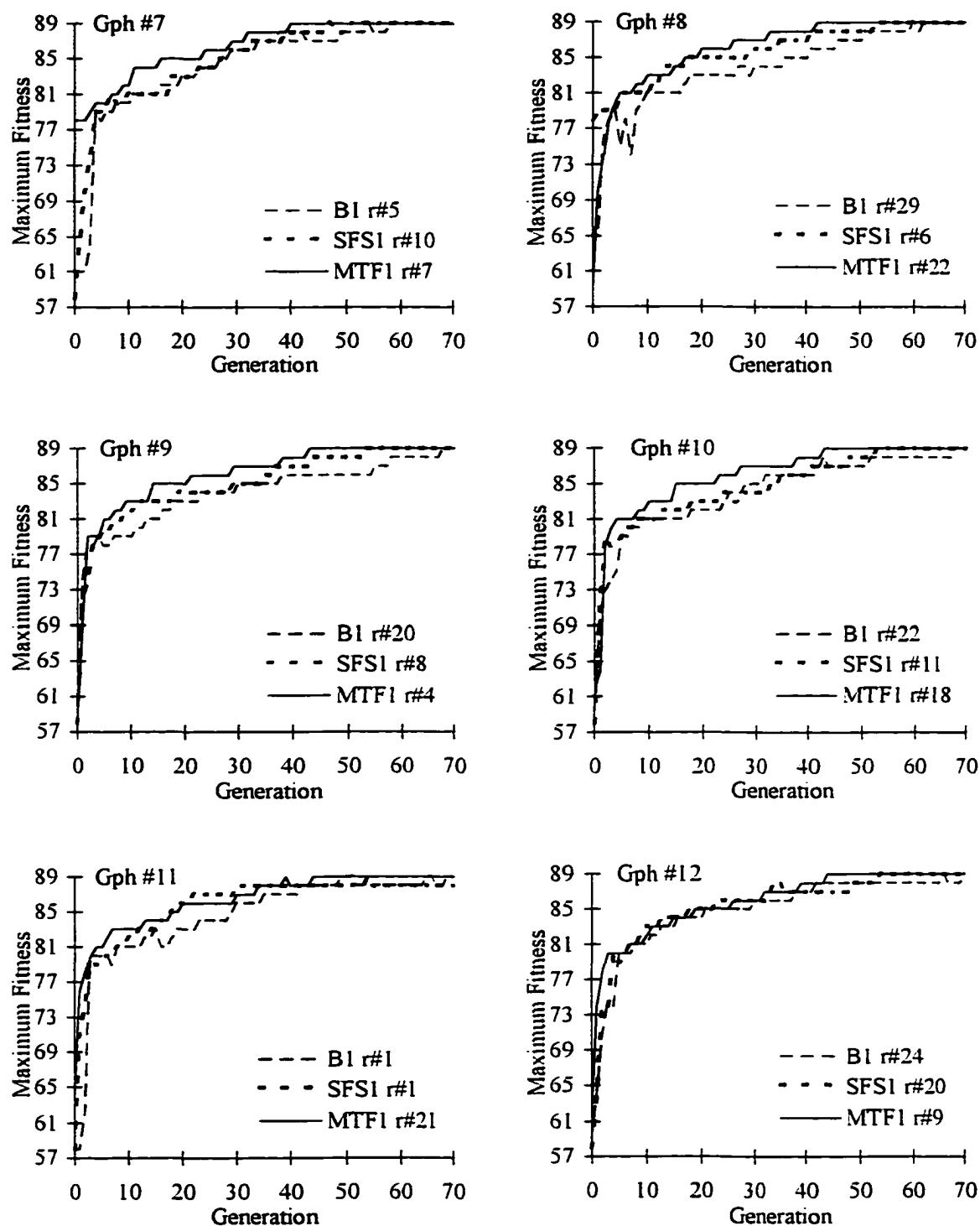


Figure C.1b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

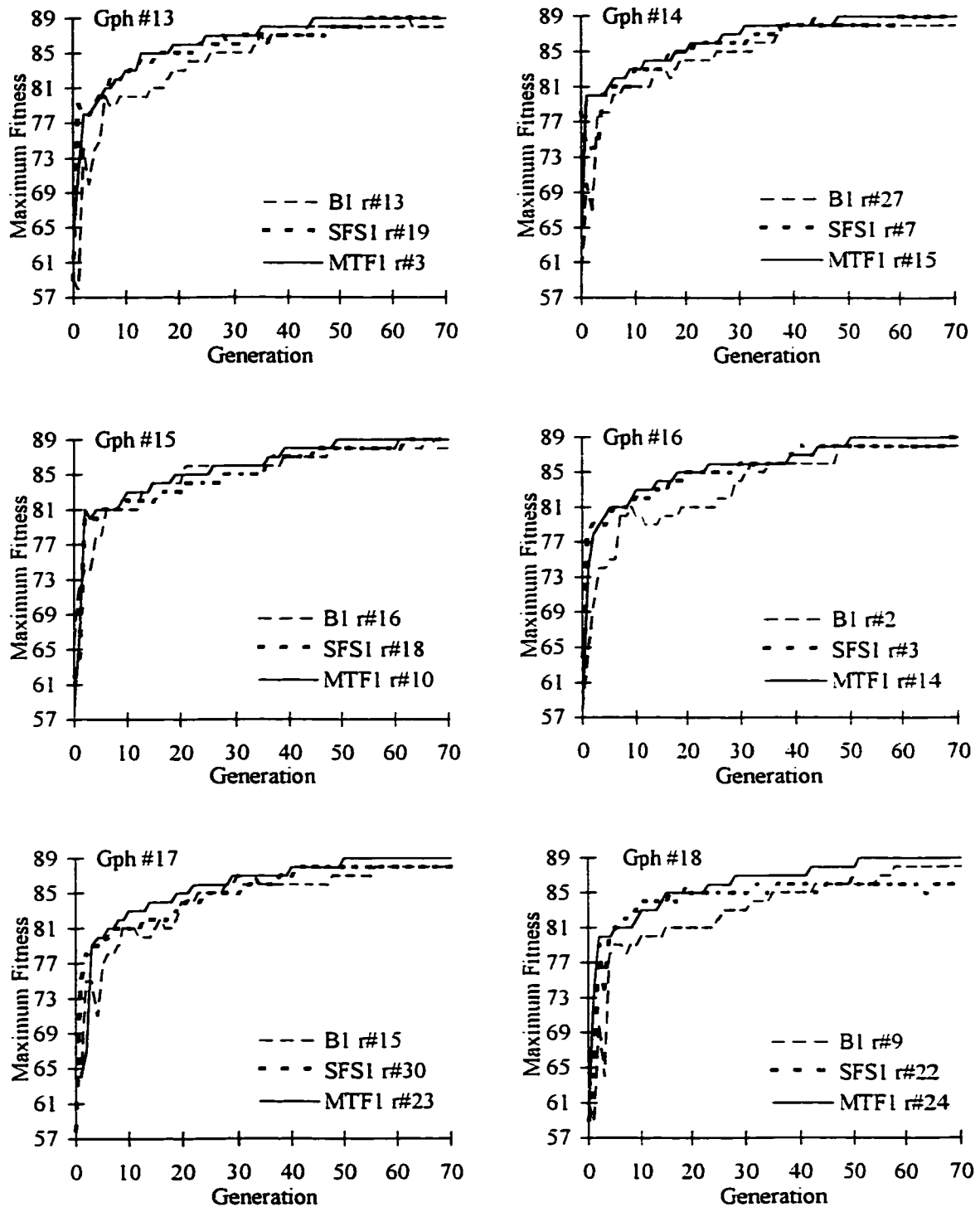


Figure C.1b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

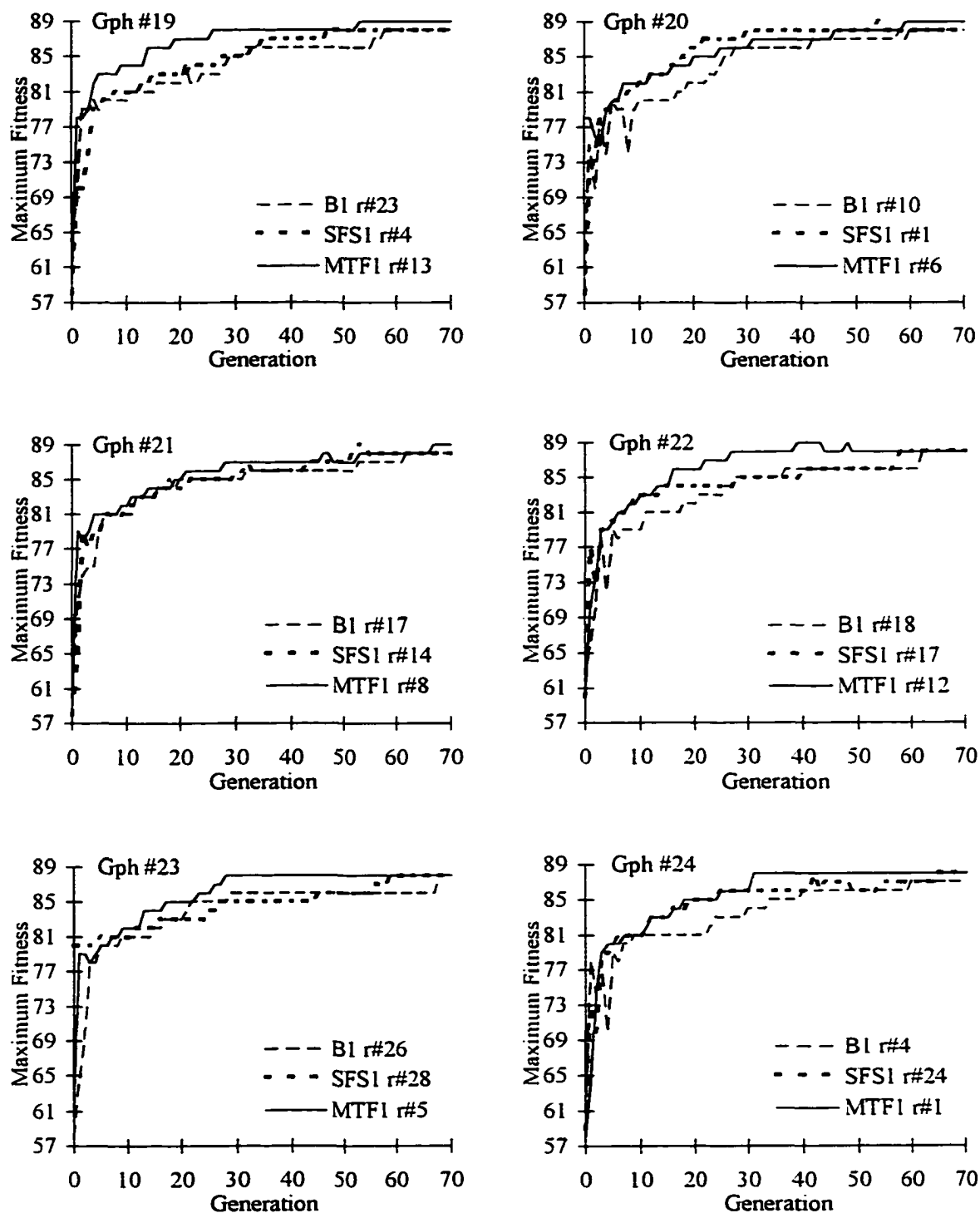


Figure C.1b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

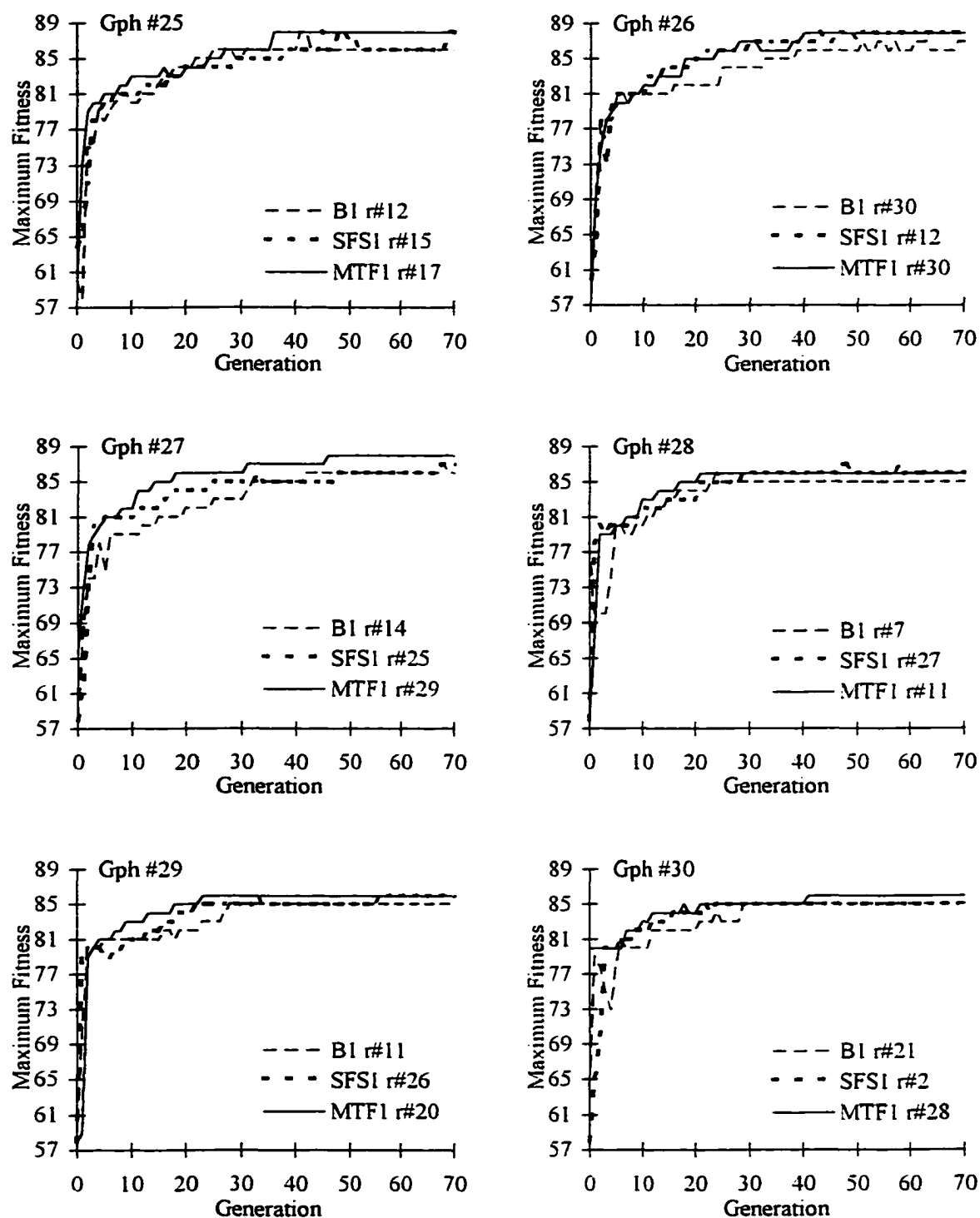


Figure C.1b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

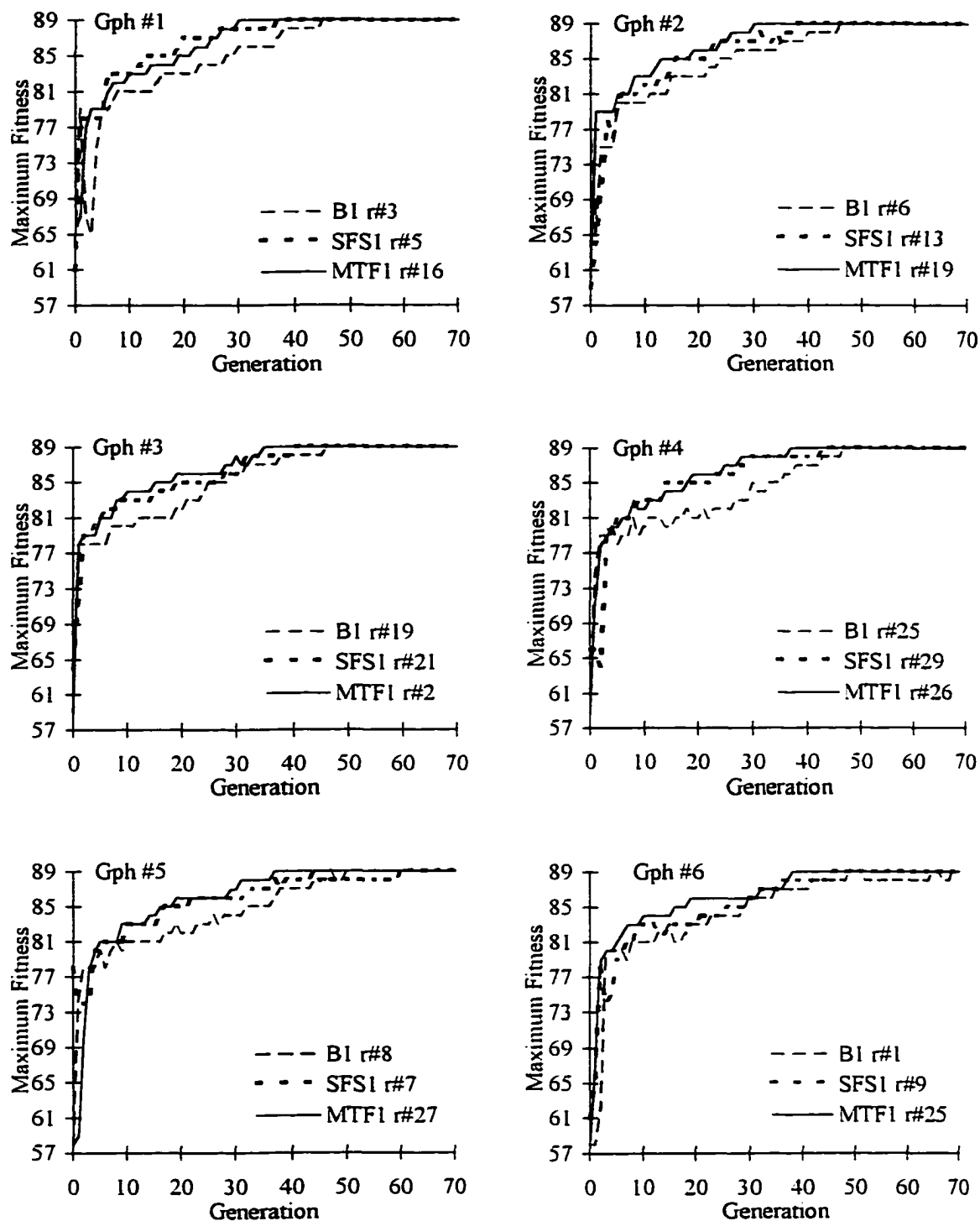


Figure C.1c: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

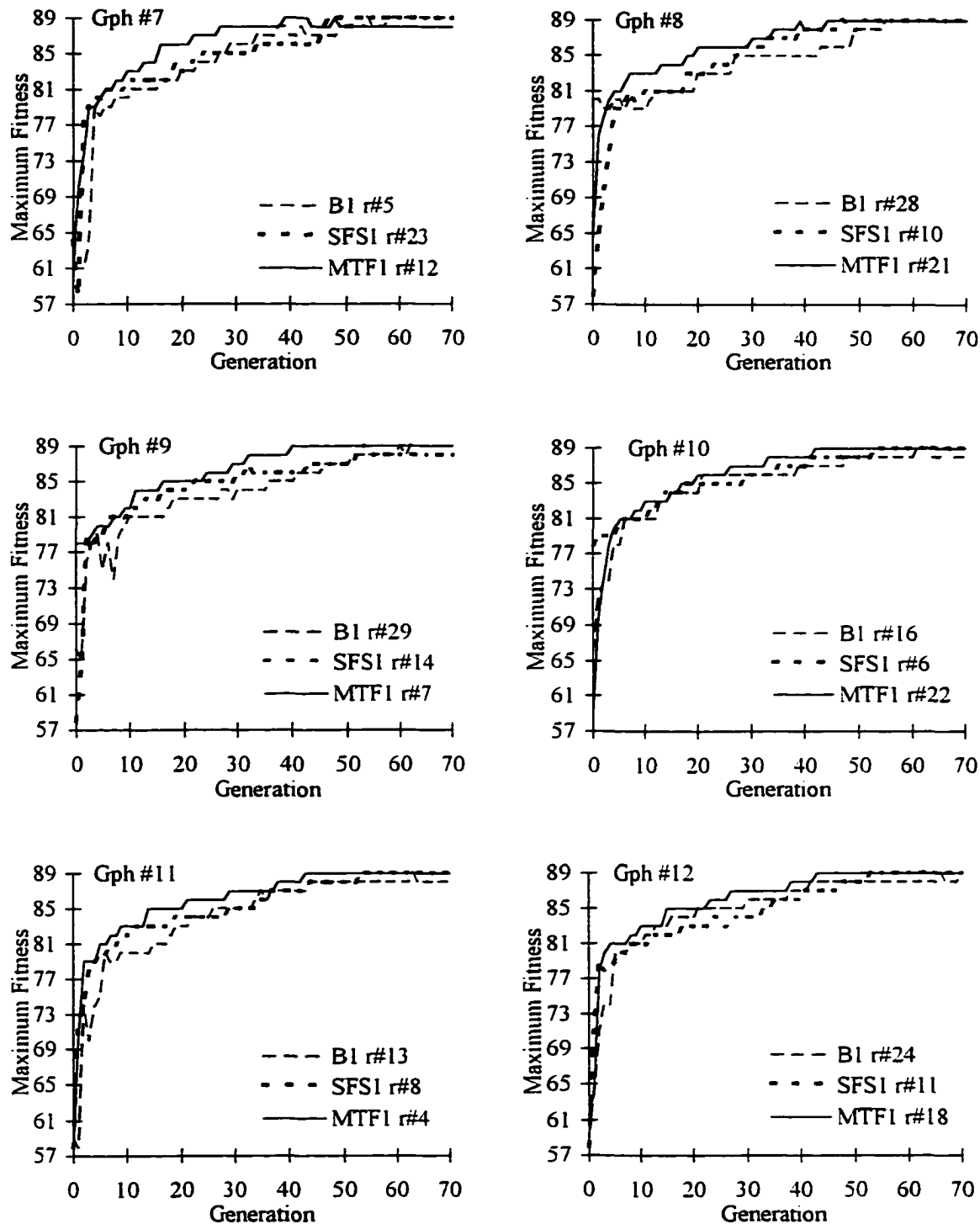


Figure C.1c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

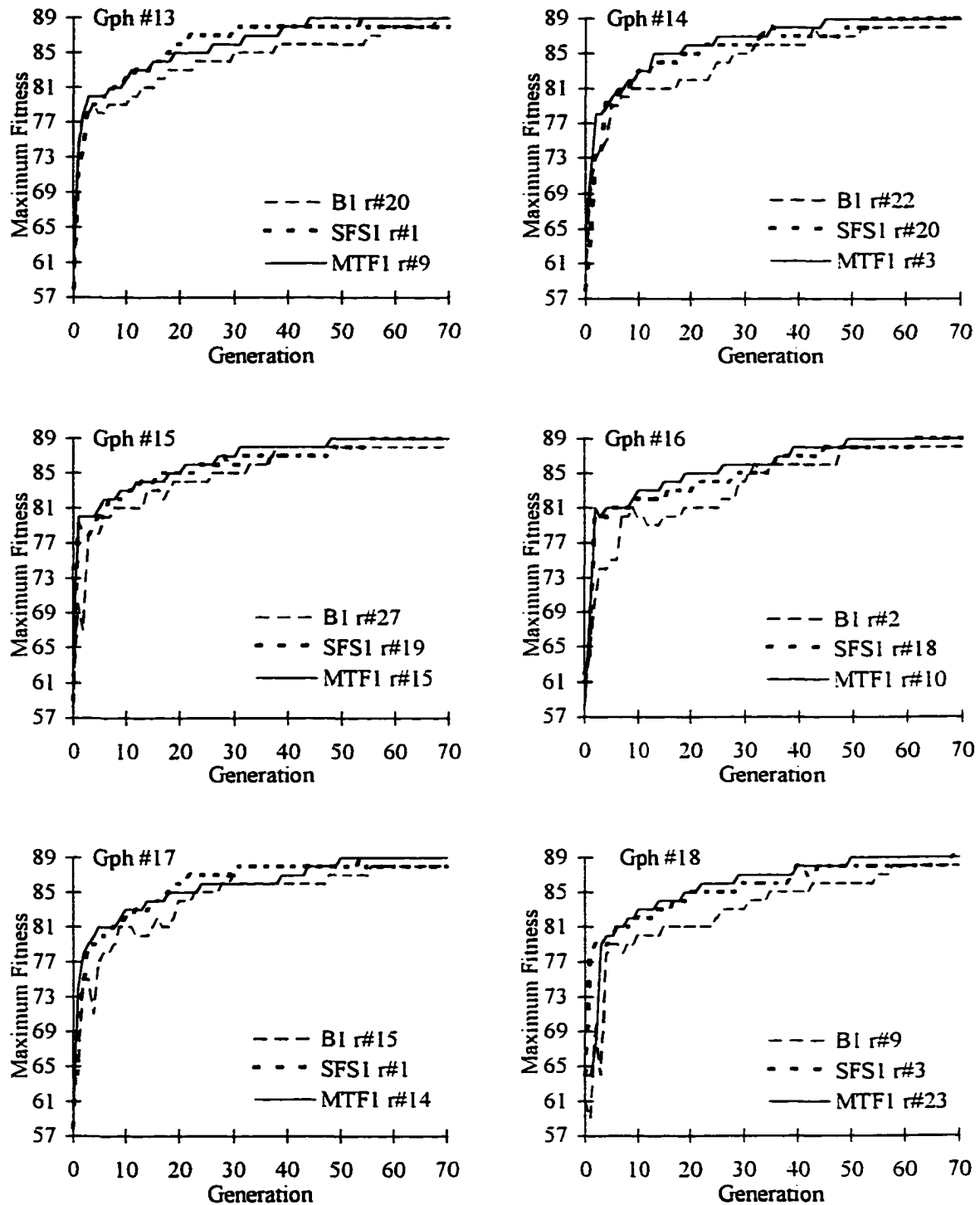


Figure C.1c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

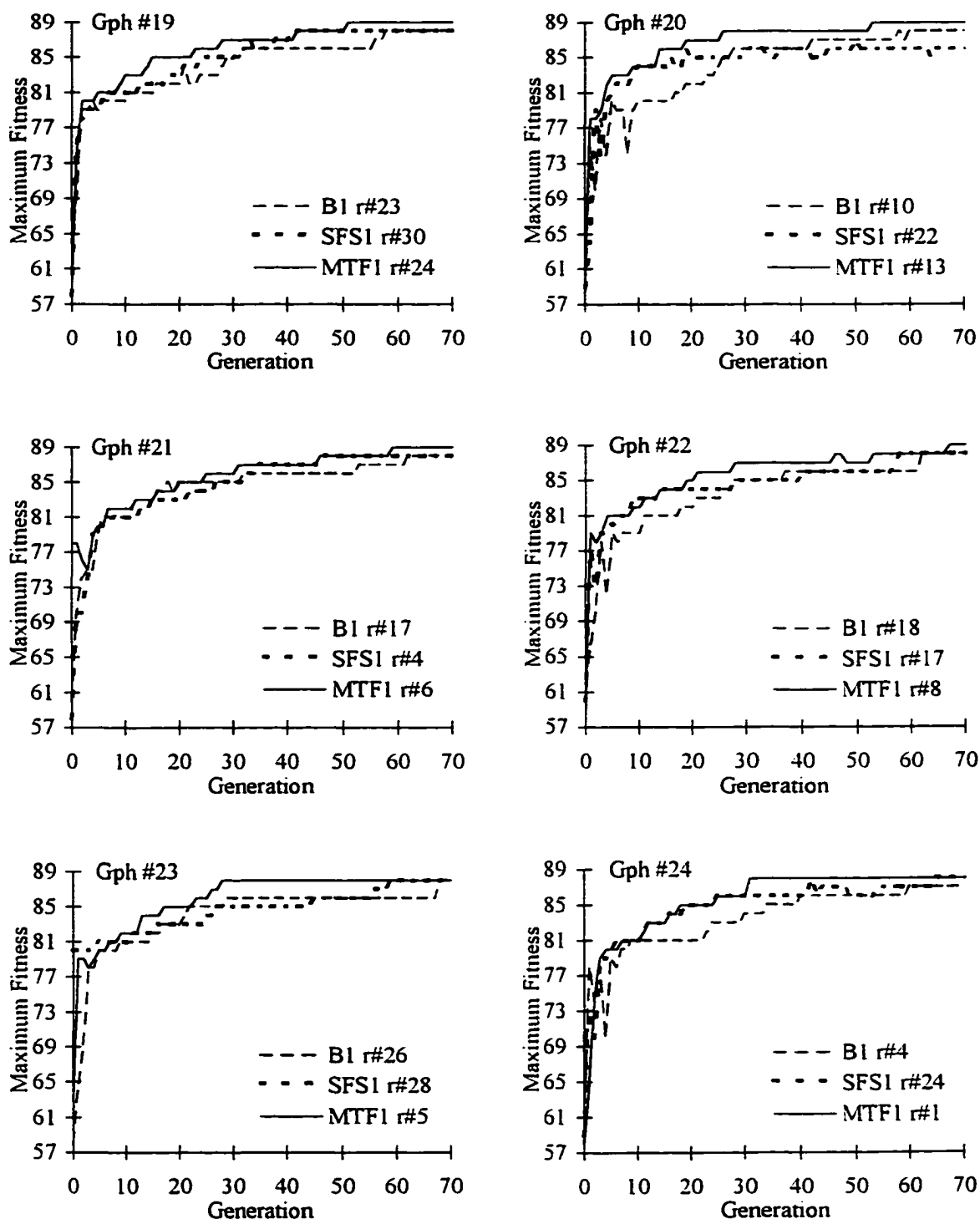


Figure C.1c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

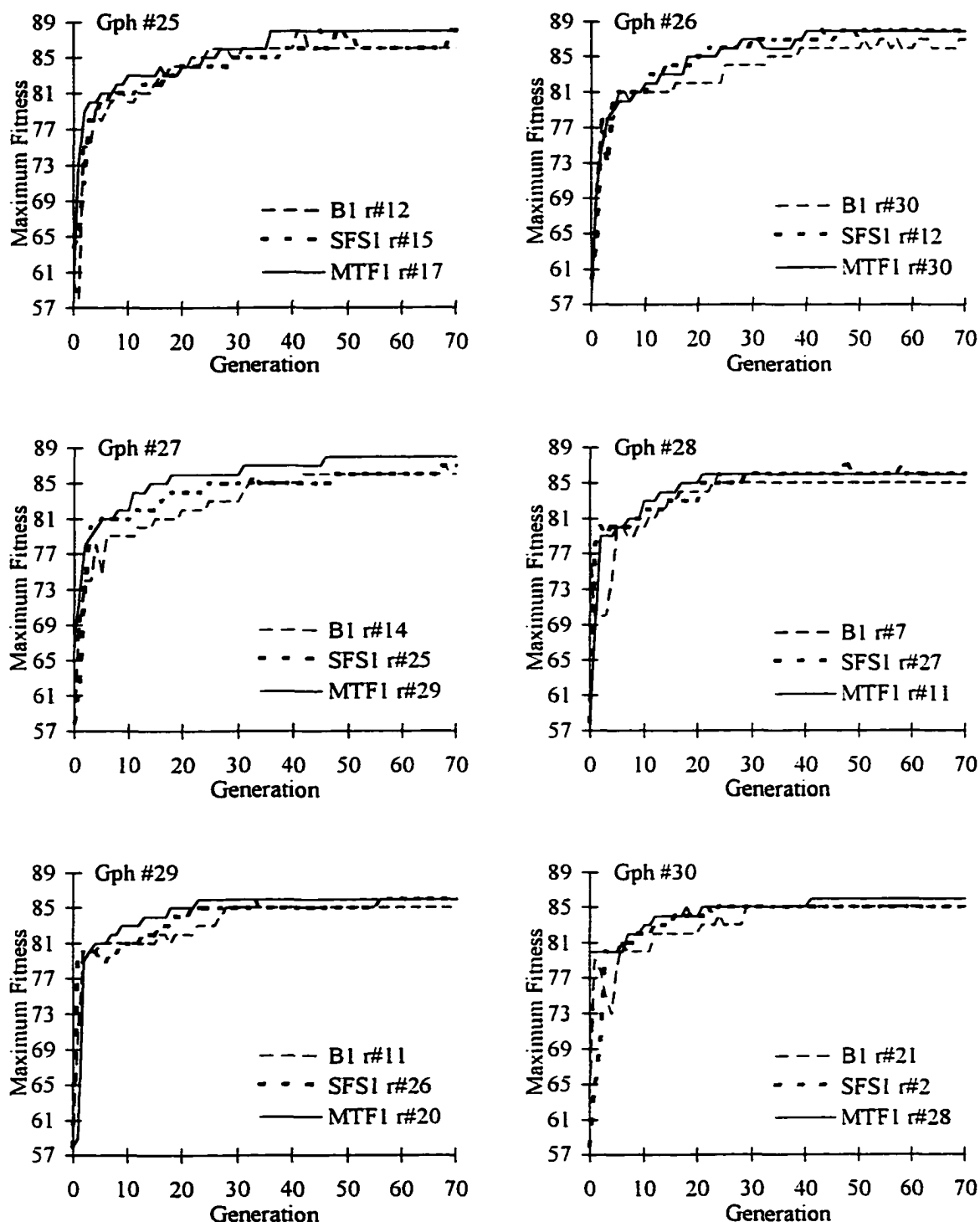


Figure C.1c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

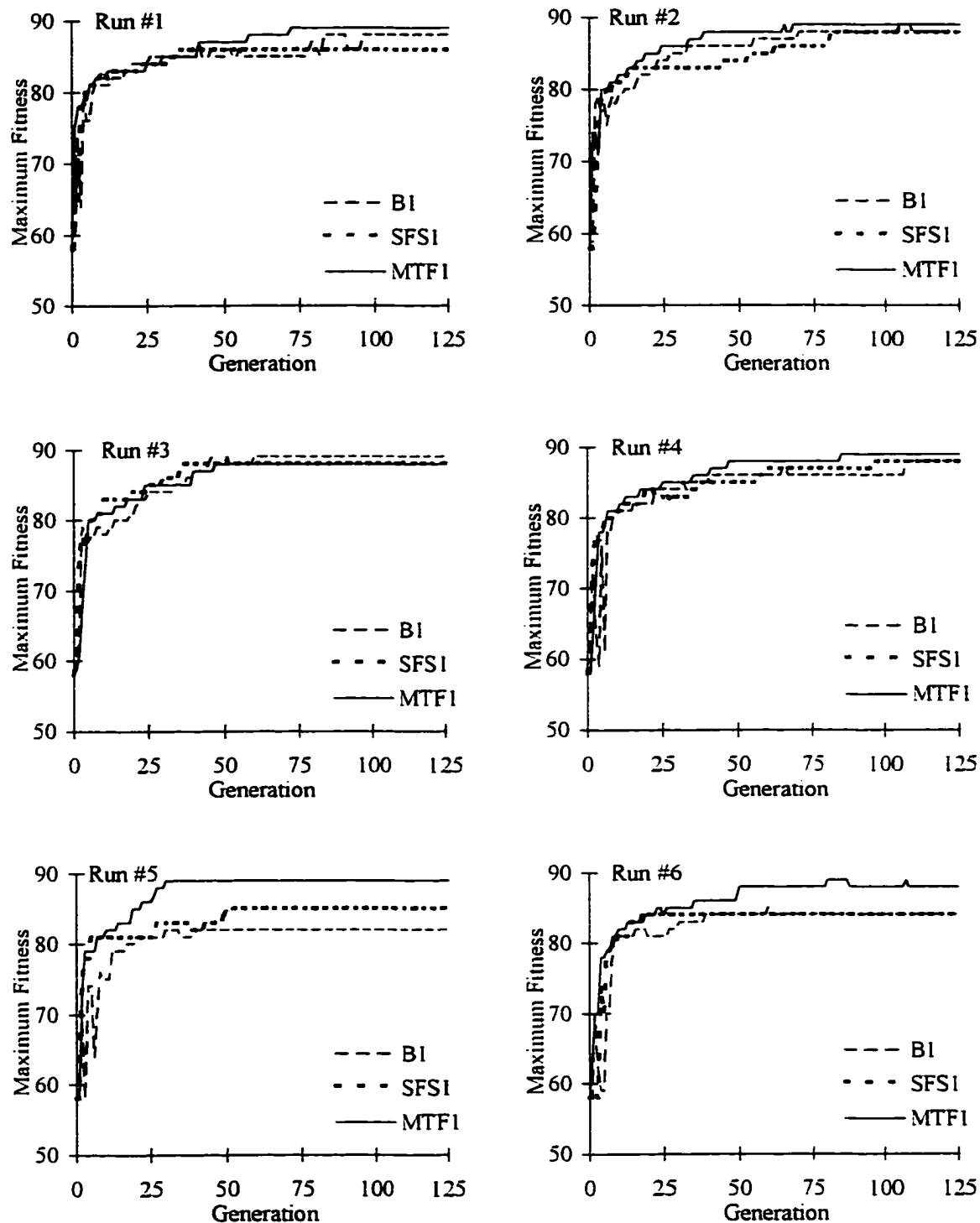


Figure C.2a: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

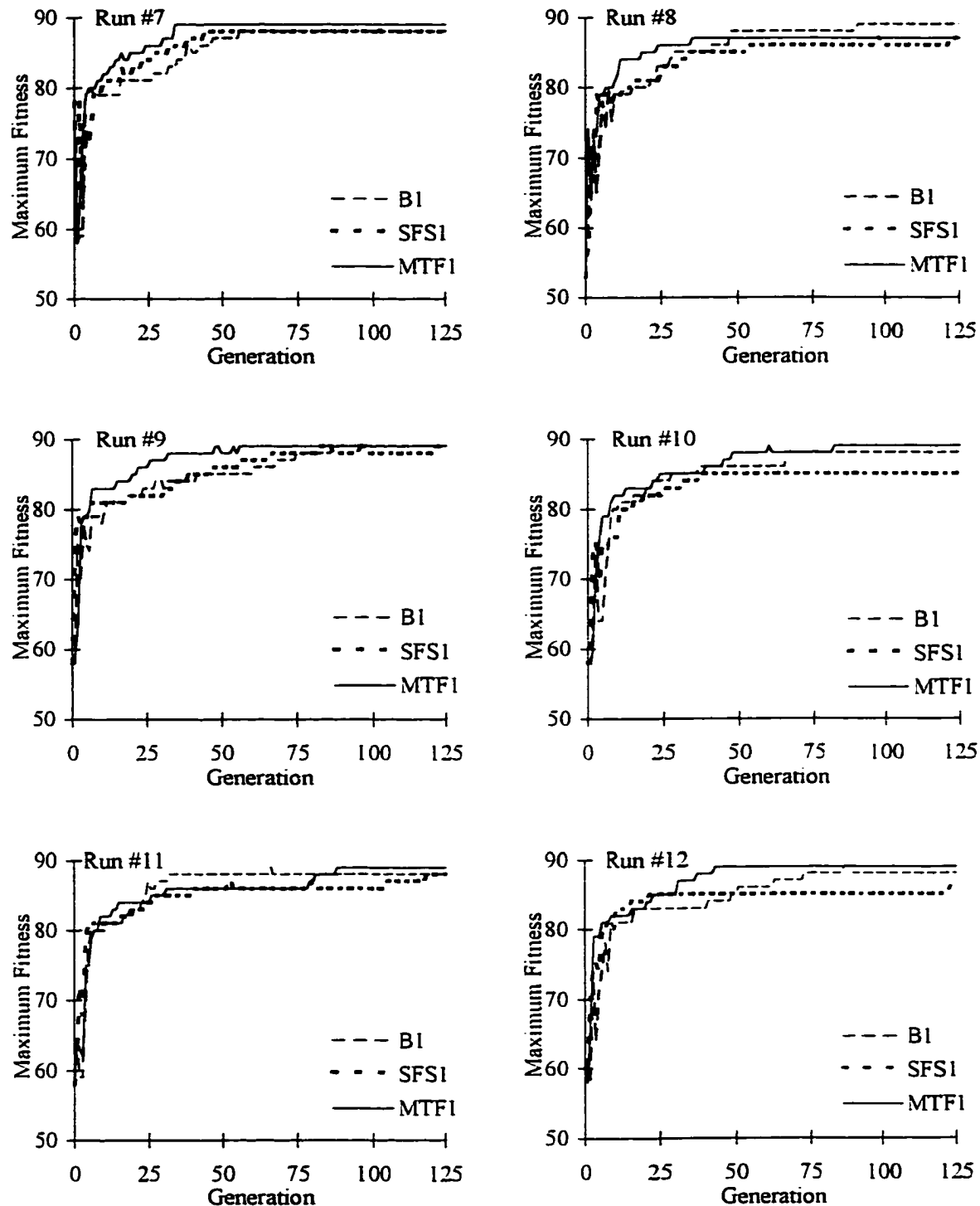


Figure C.2a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

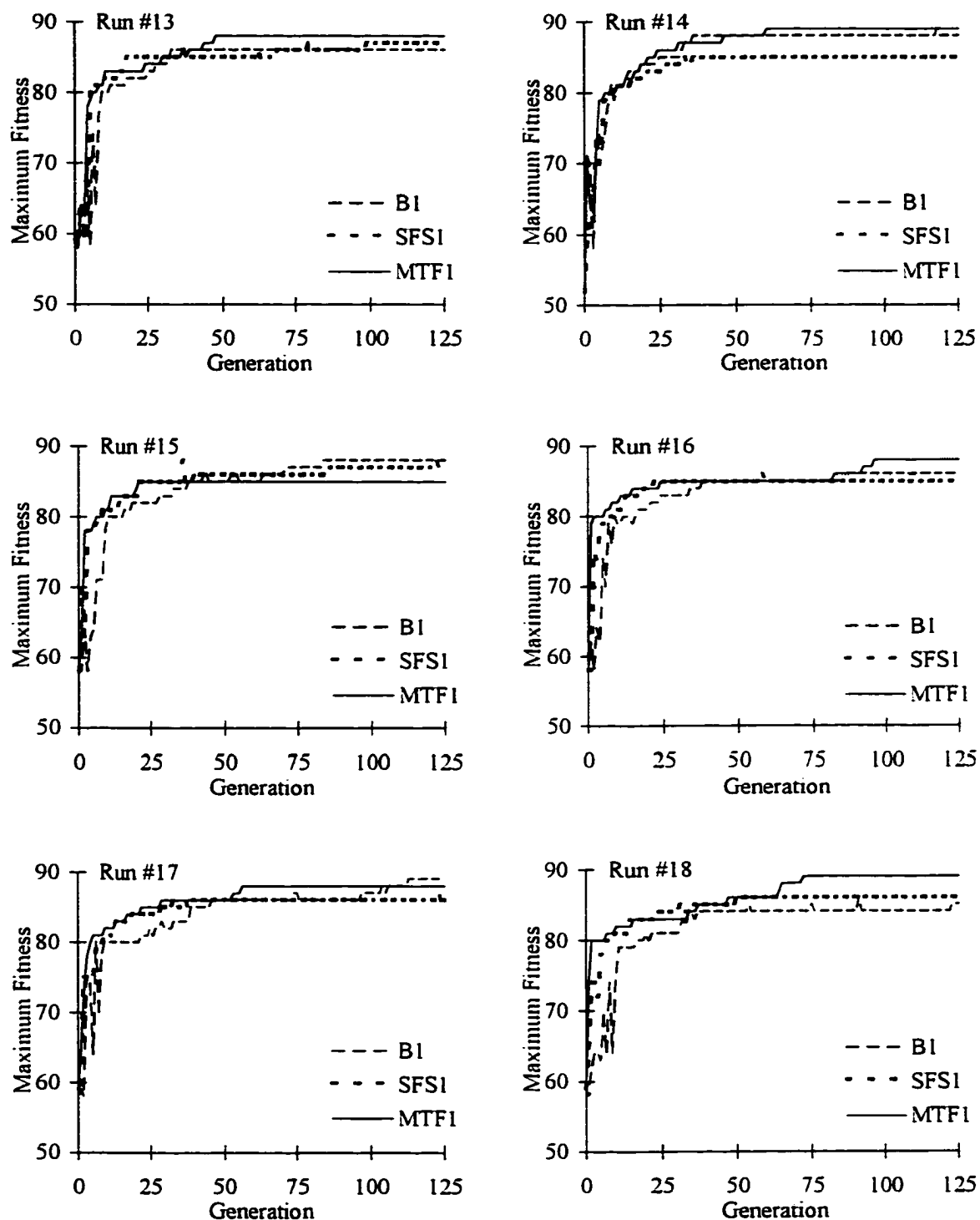


Figure C.2a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

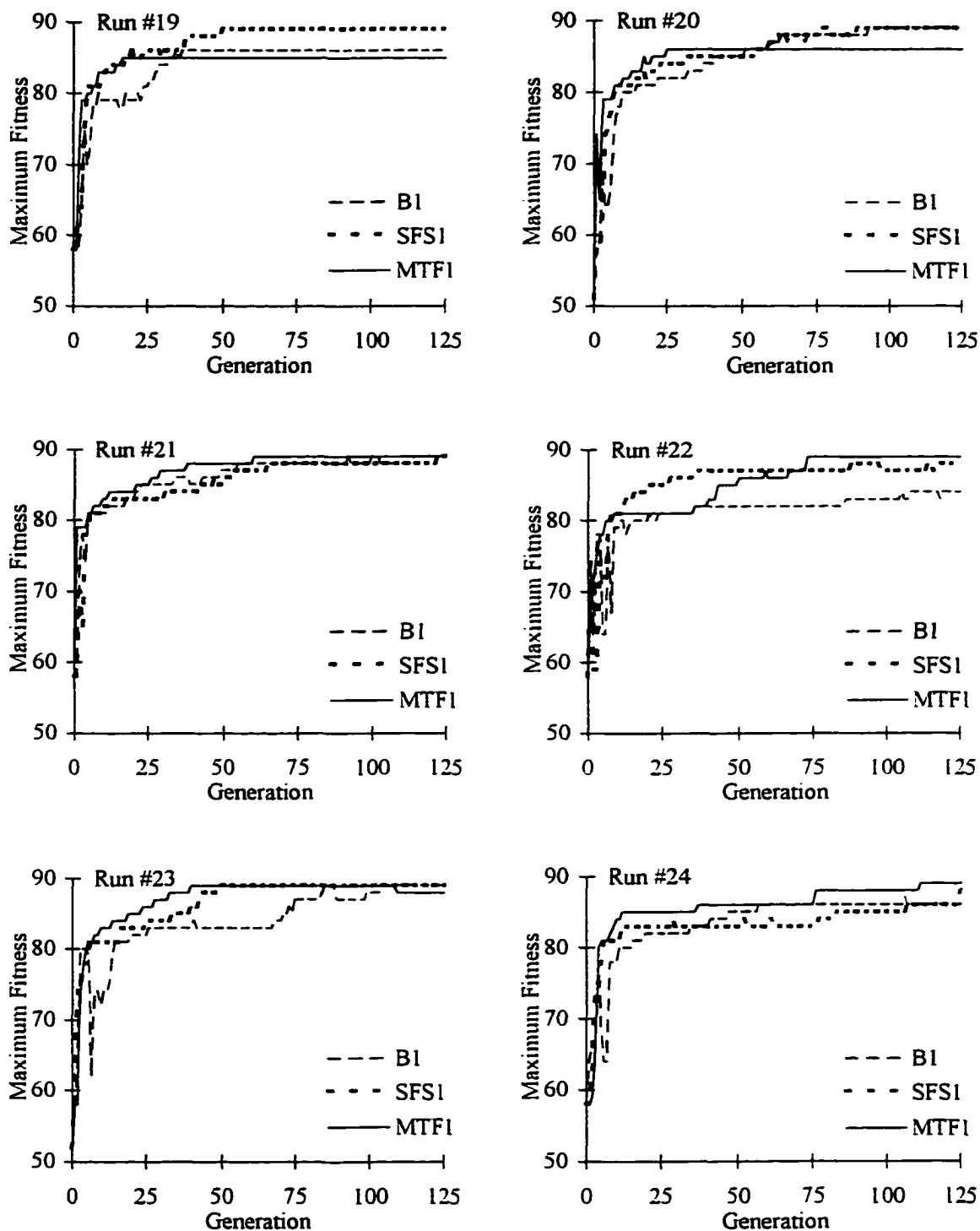


Figure C.2a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

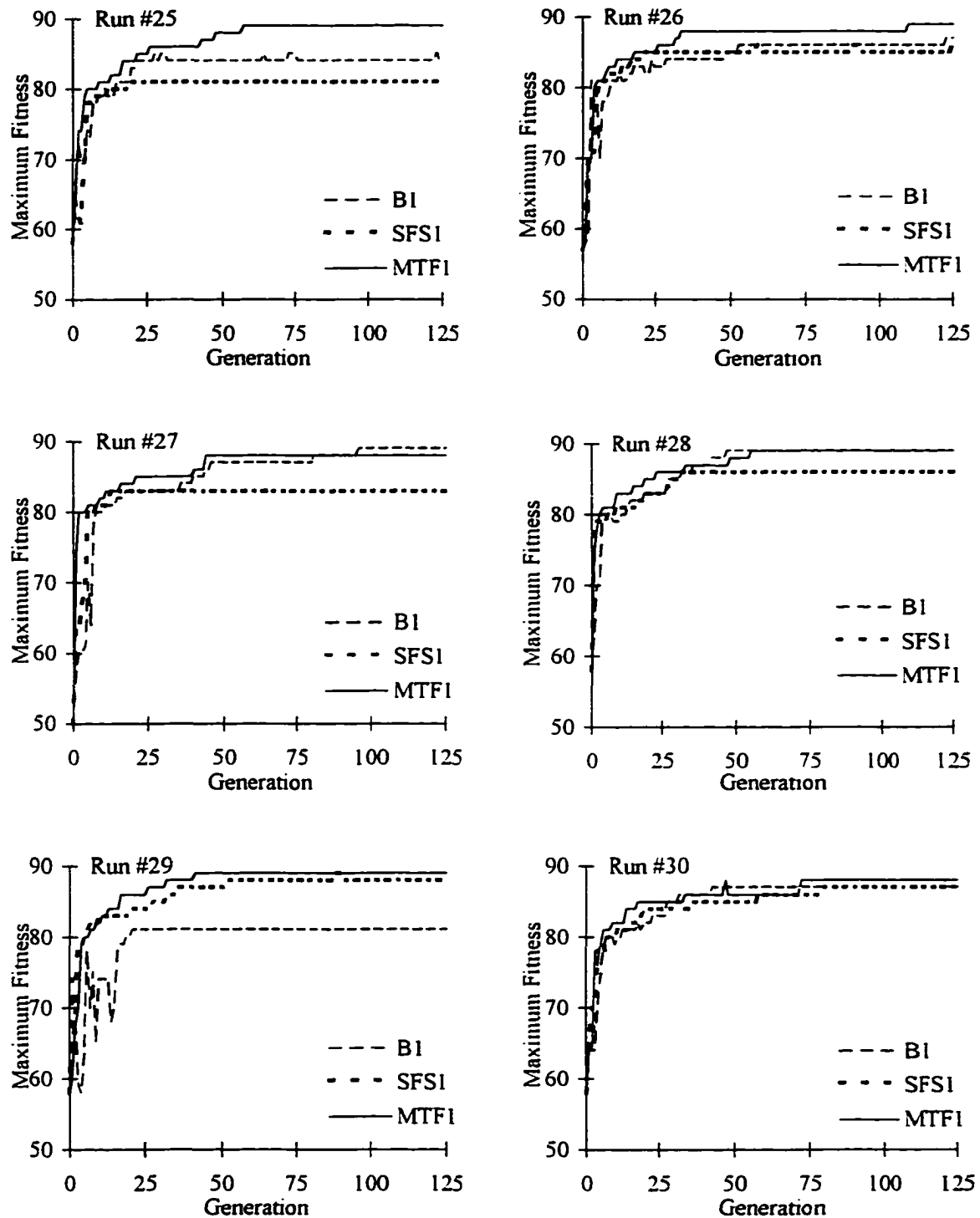


Figure C.2a continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

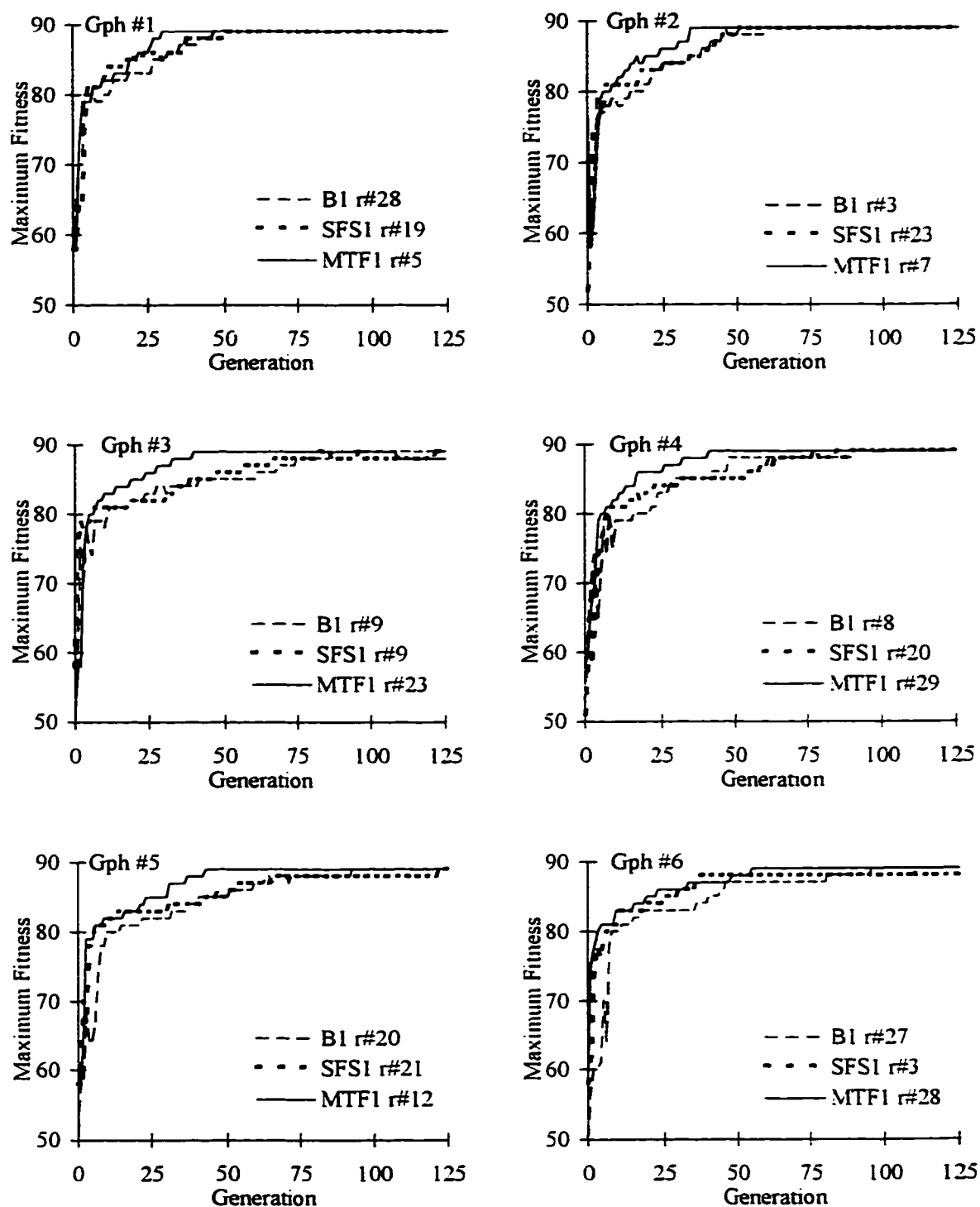


Figure C.2b: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

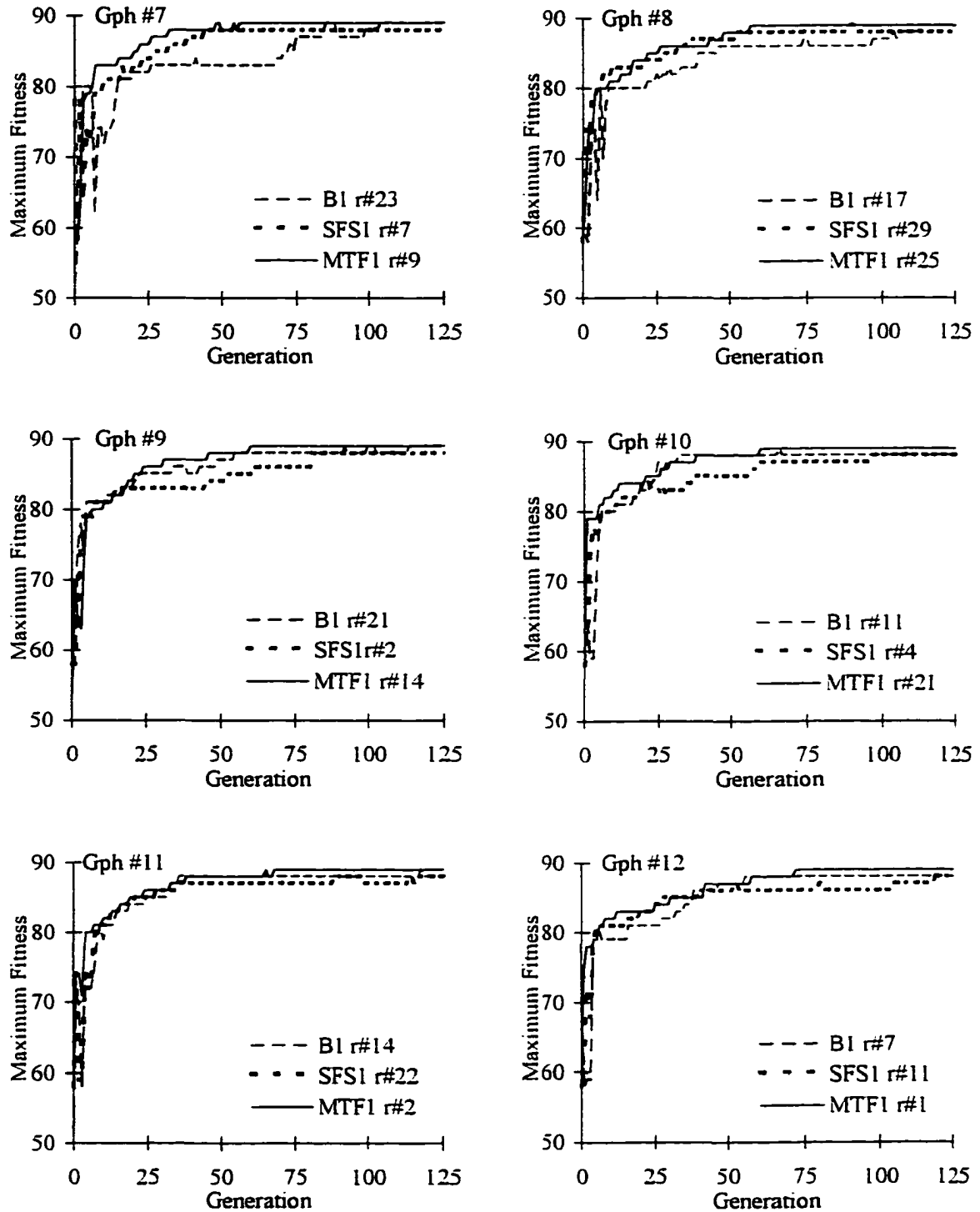


Figure C.2b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

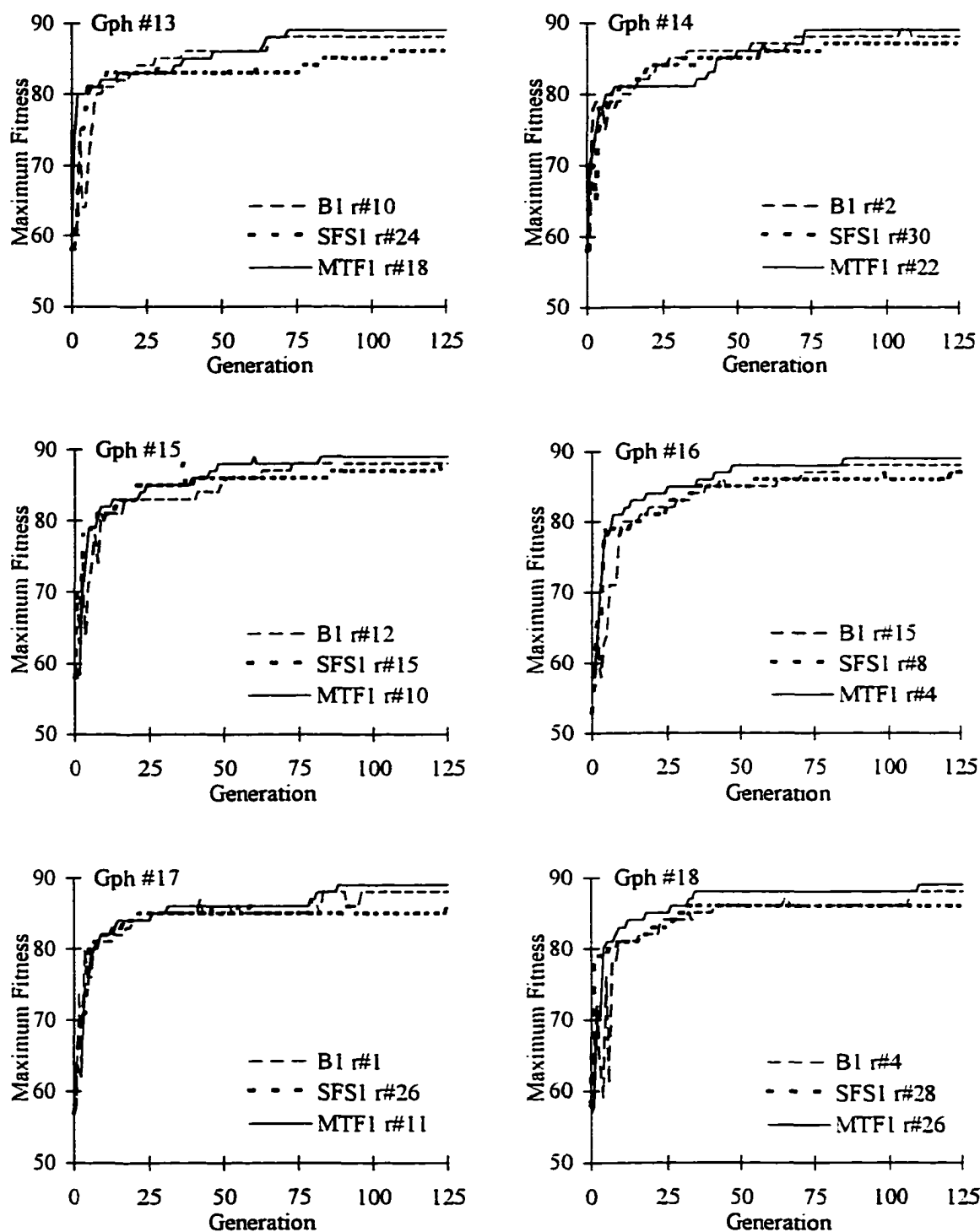


Figure C.2b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

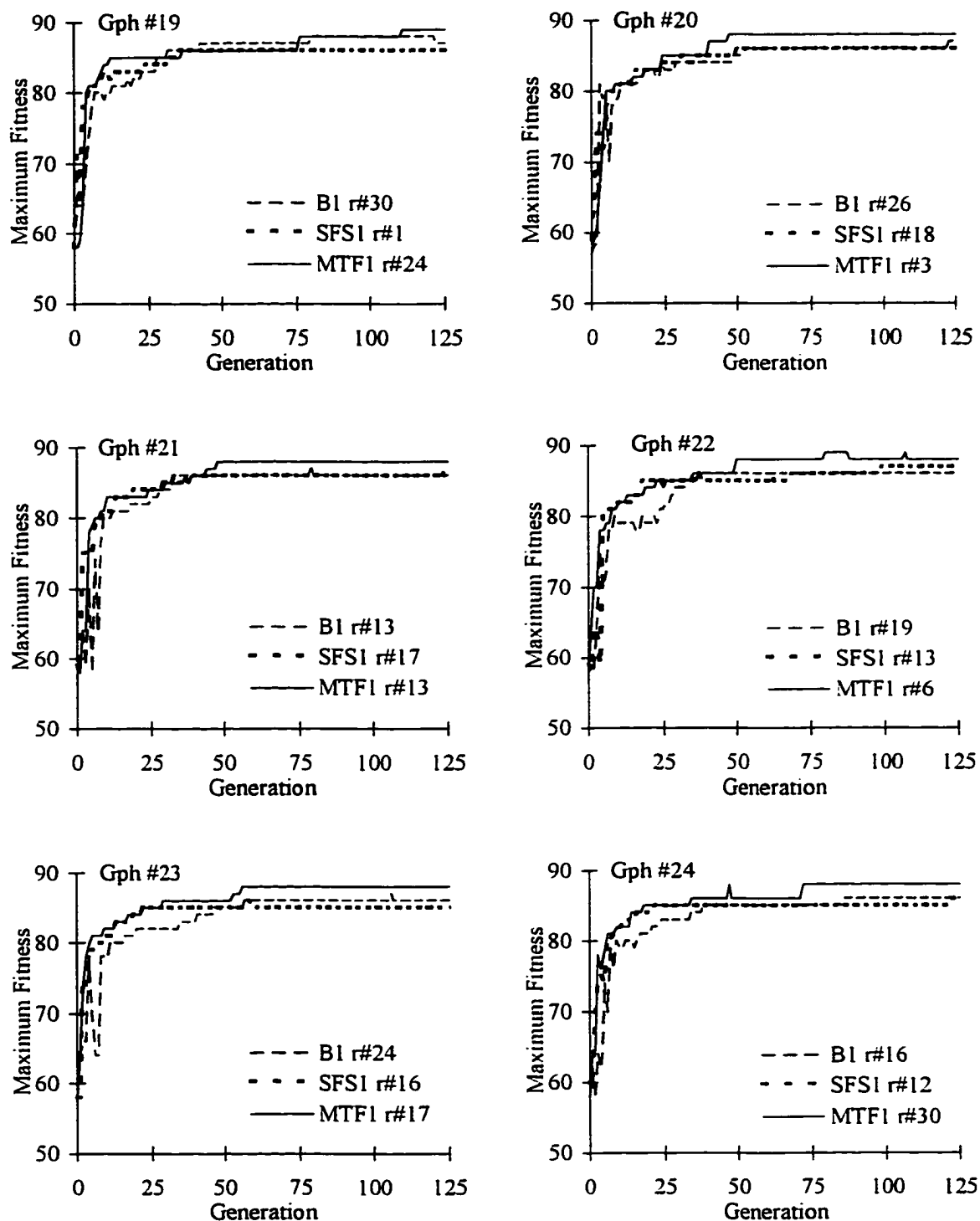


Figure C.2b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Evolutionary progress population is basis for comparison.

Appendix C continued

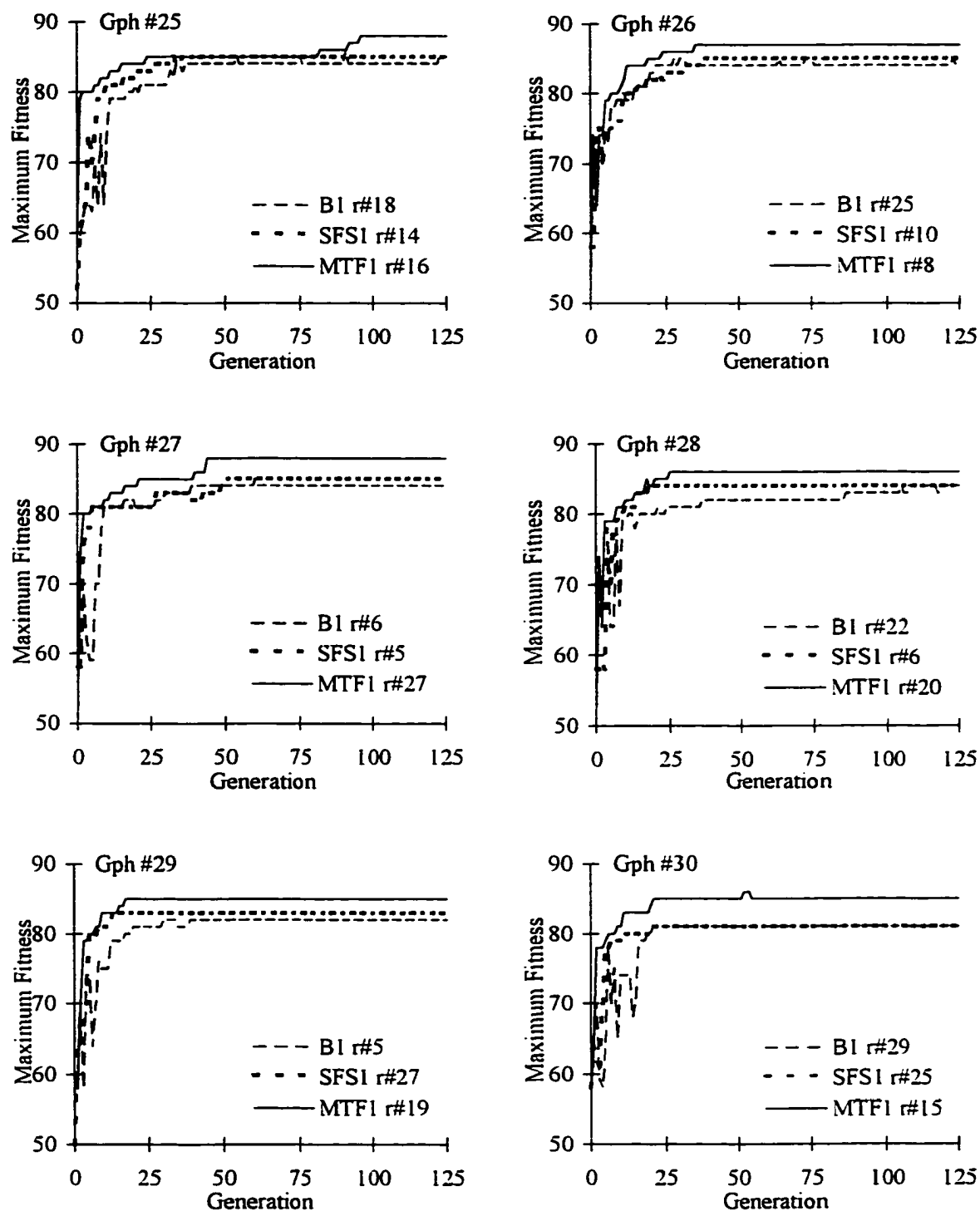


Figure C.2b continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Evolutionary progress population is basis for comparison.

Appendix C continued

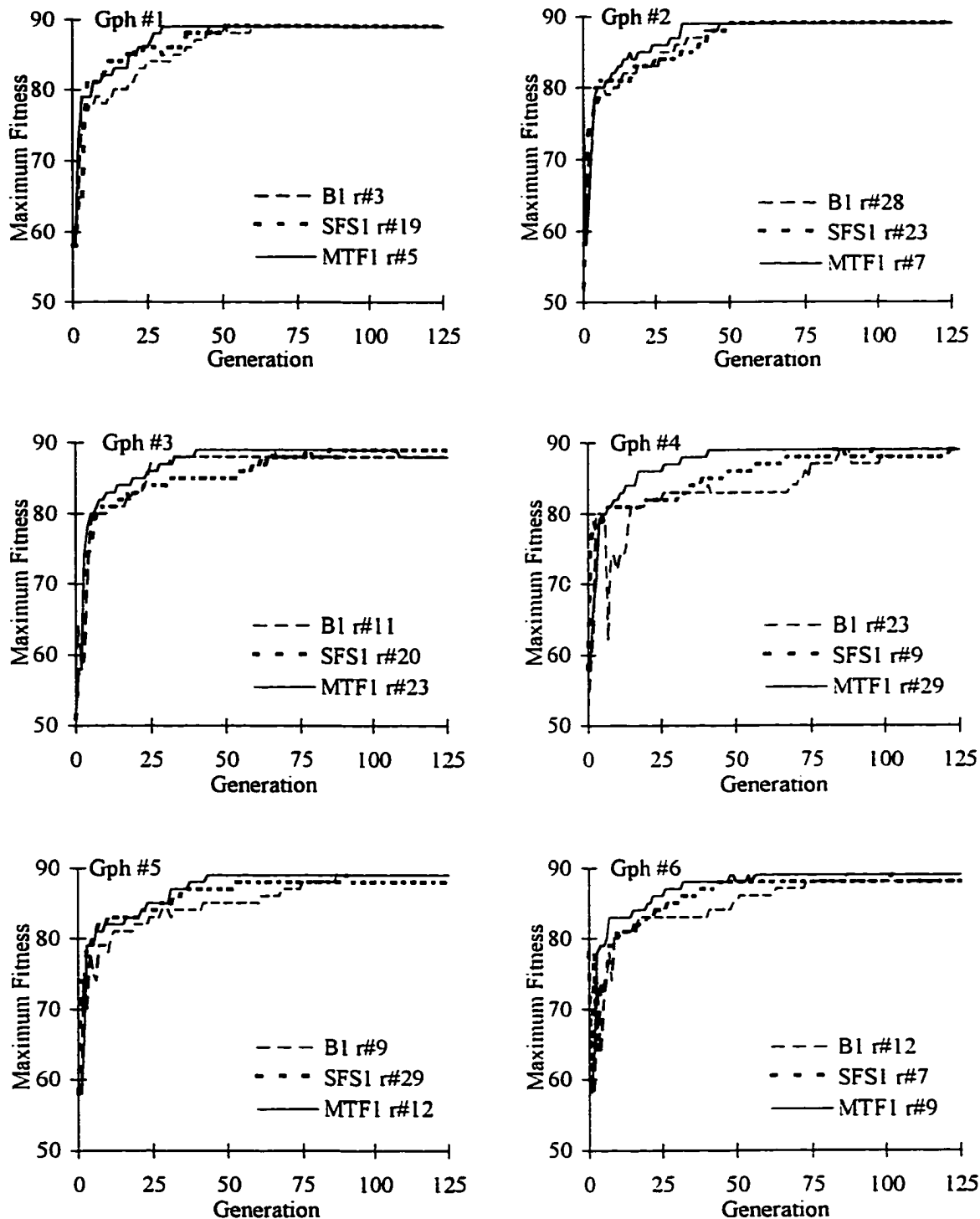


Figure C.2c: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

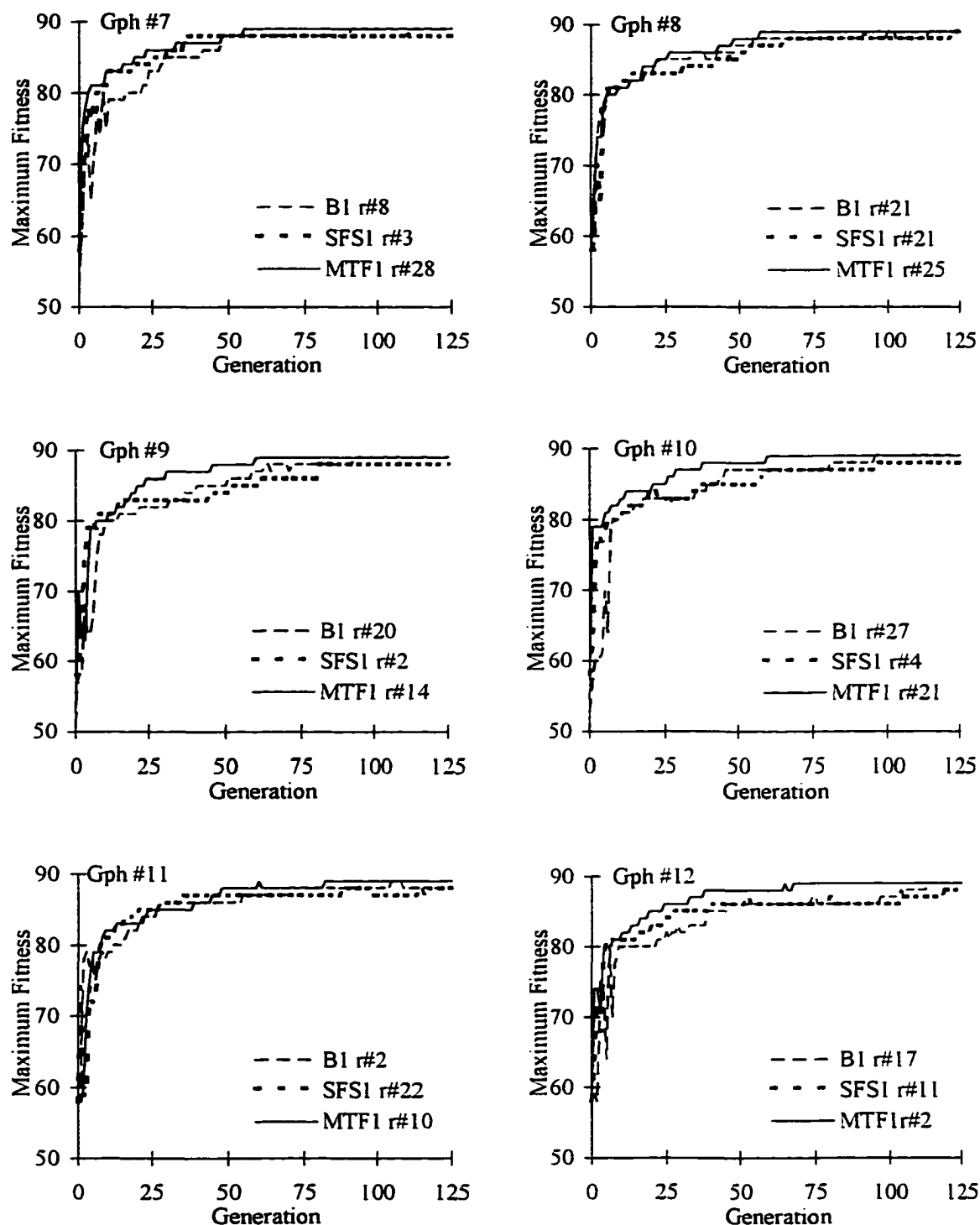


Figure C.2c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

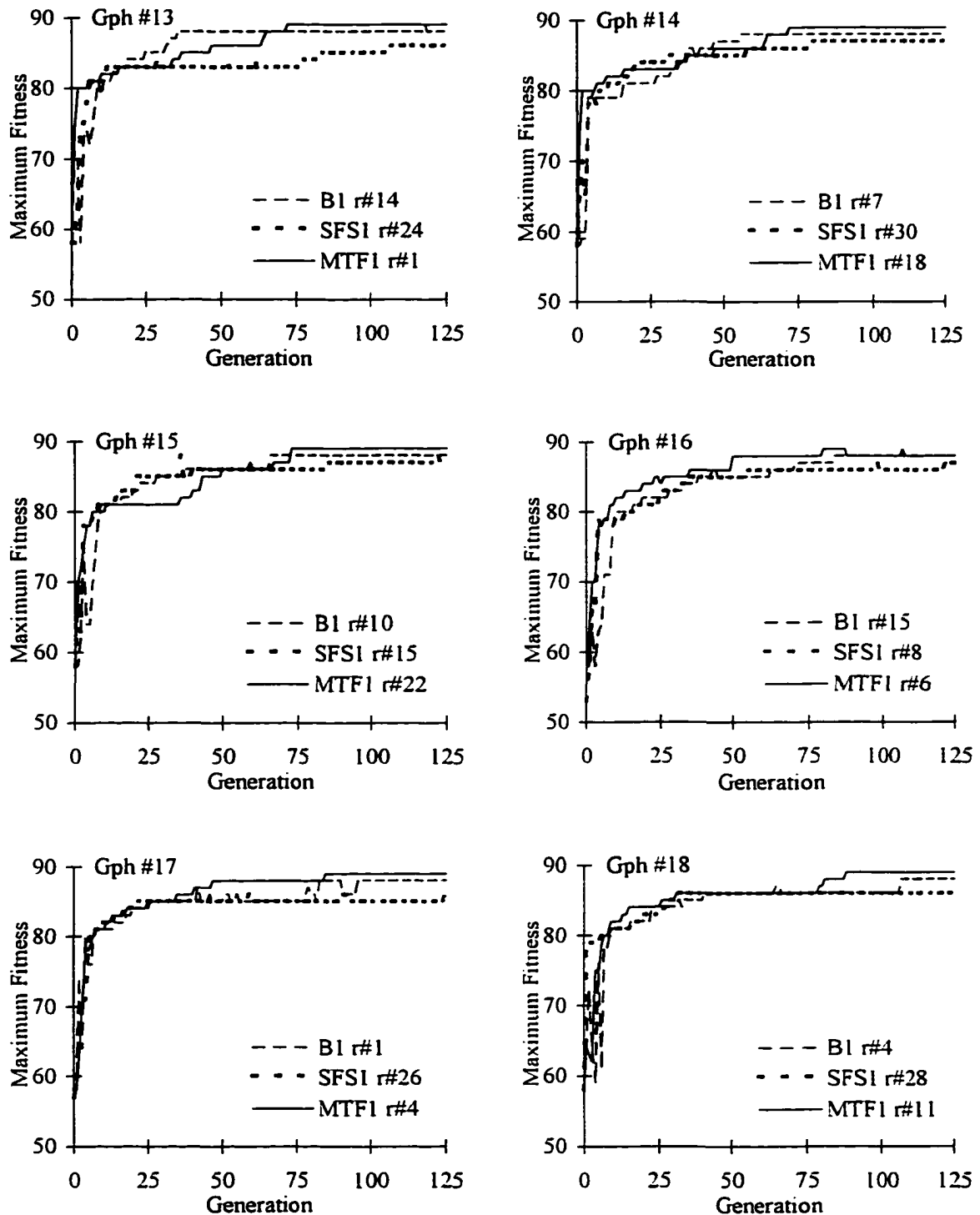


Figure C.2c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

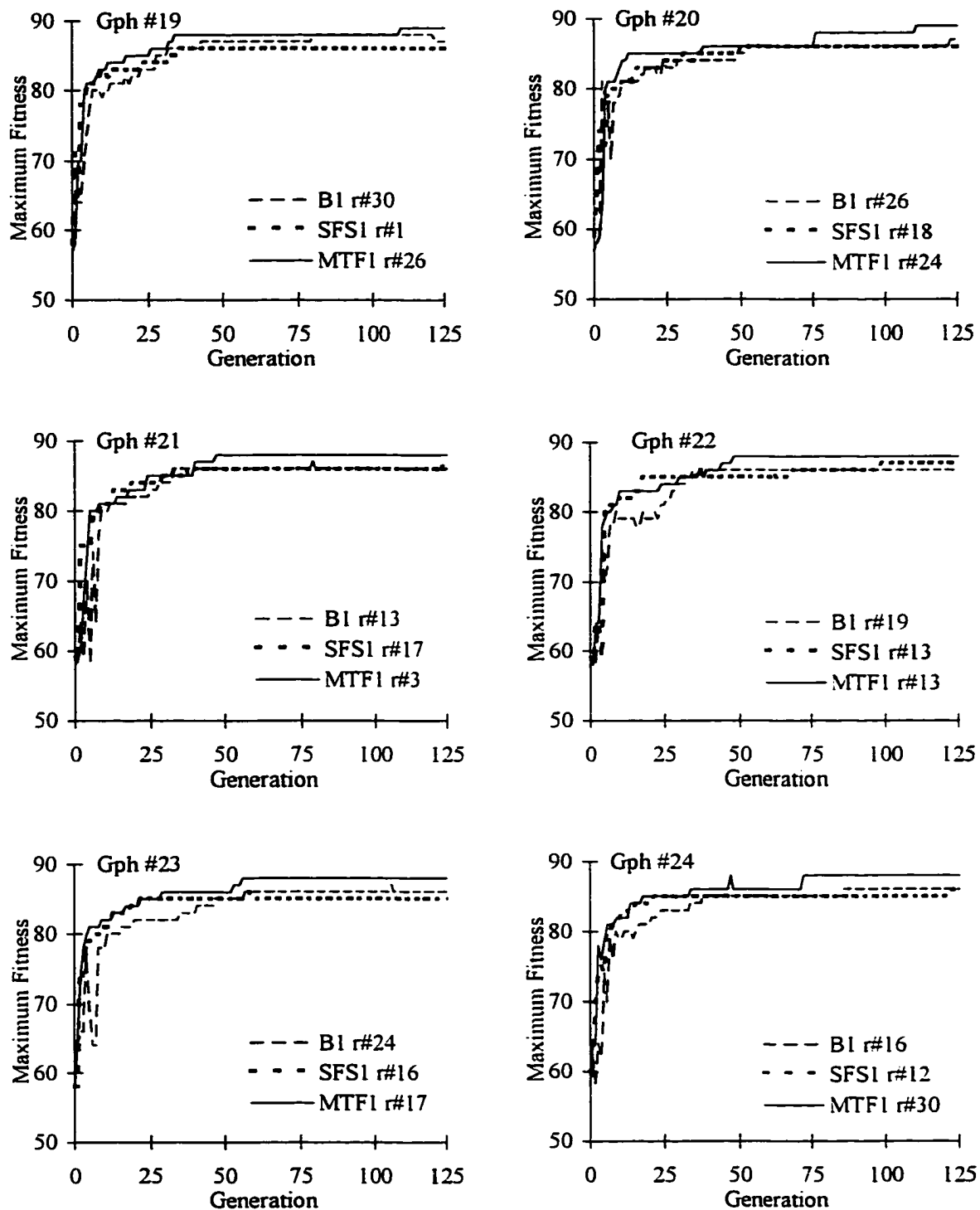


Figure C.2c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

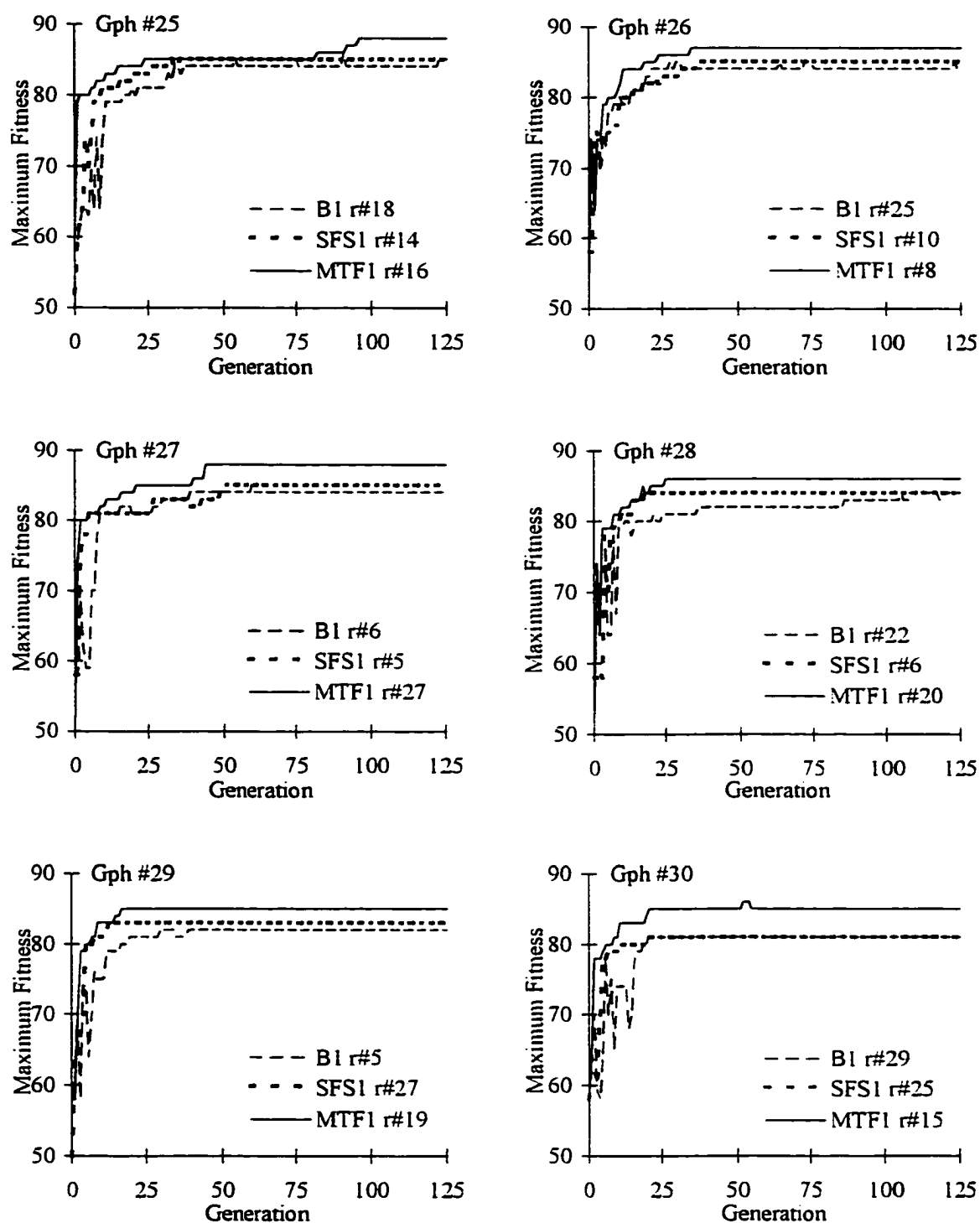


Figure C.2c continued: Comparison of benchmark (B1), benchmark with the SFS algorithm (SFS1), and benchmark with the MTF algorithm (MTF1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

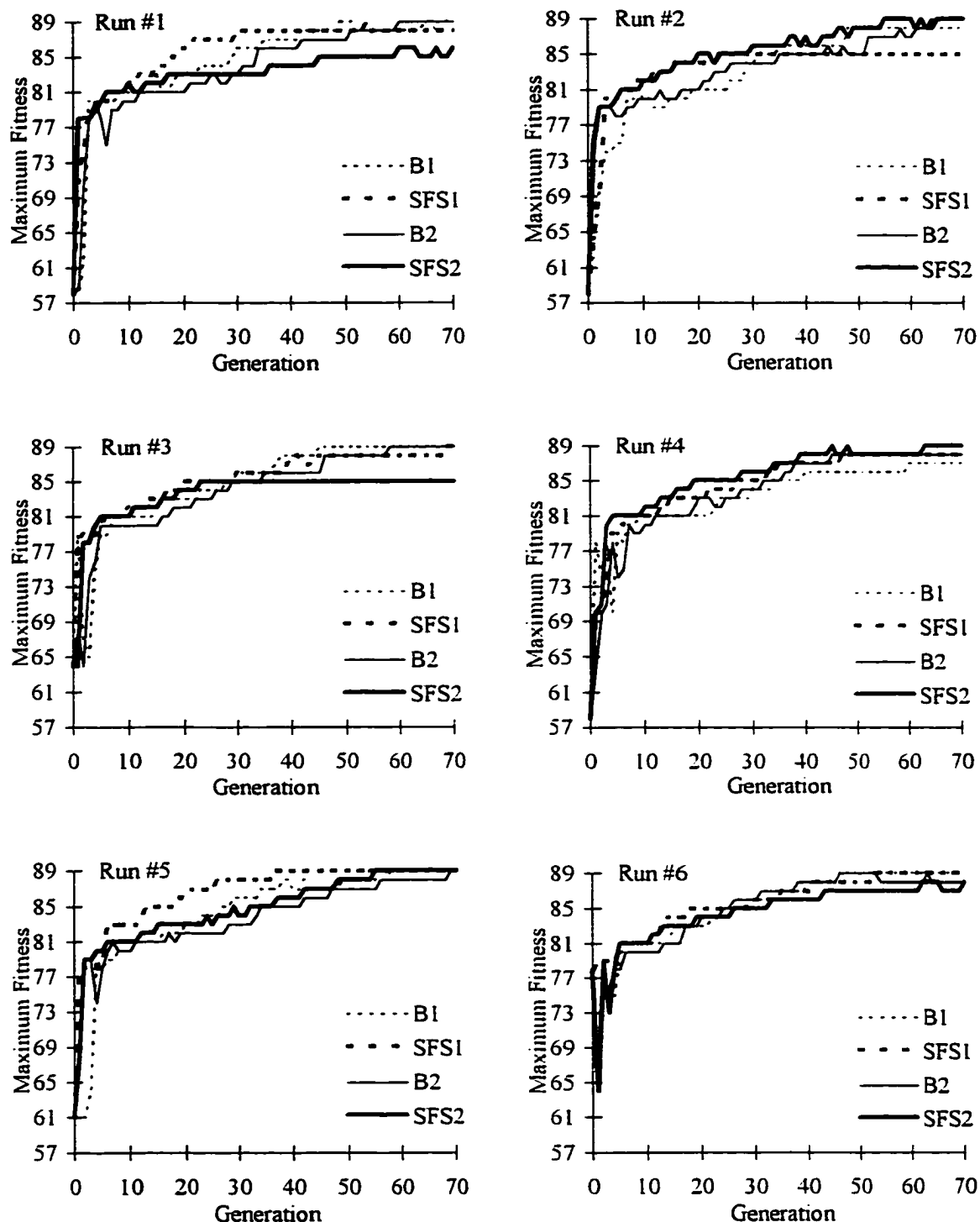


Figure C.3a: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

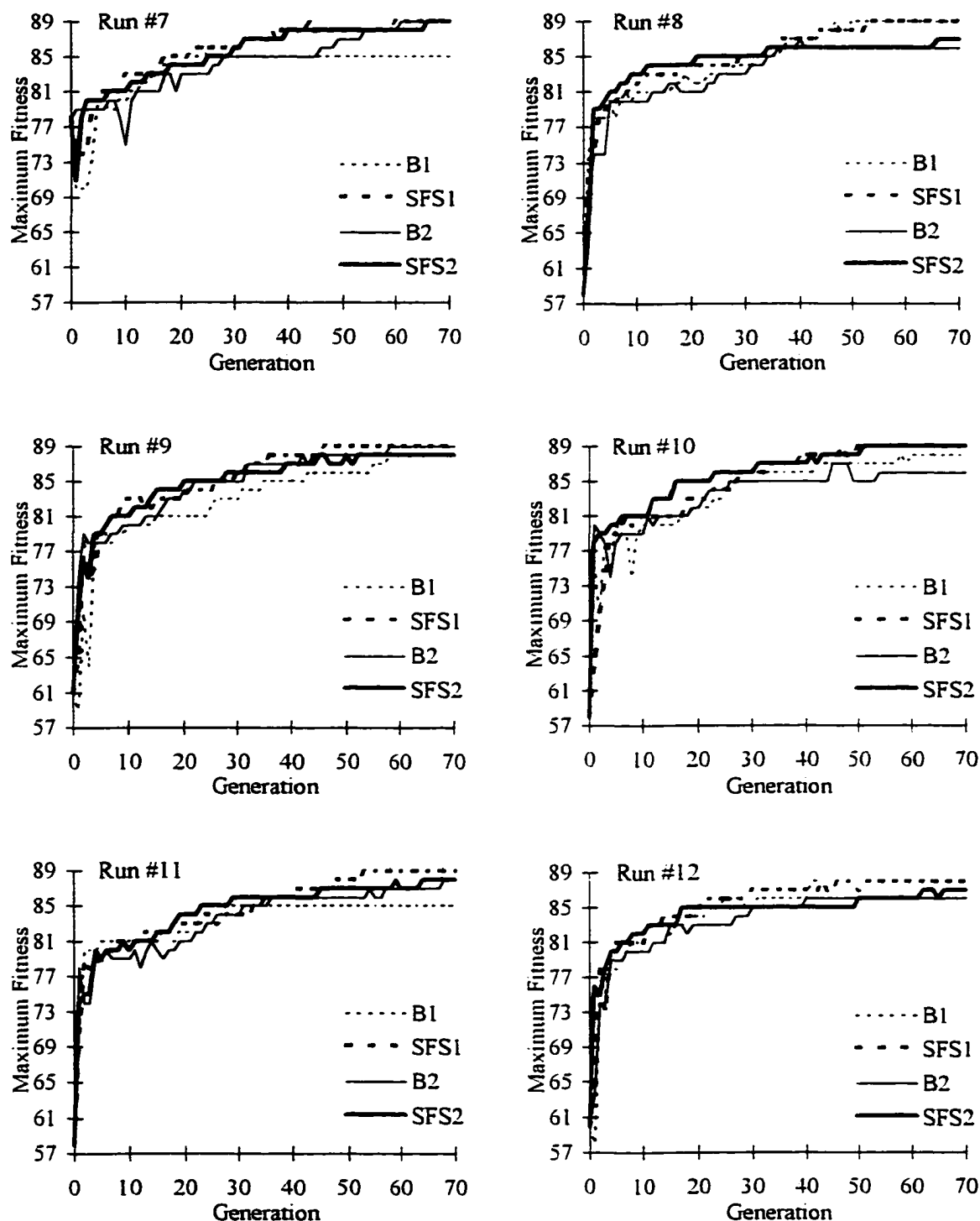


Figure C.3a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

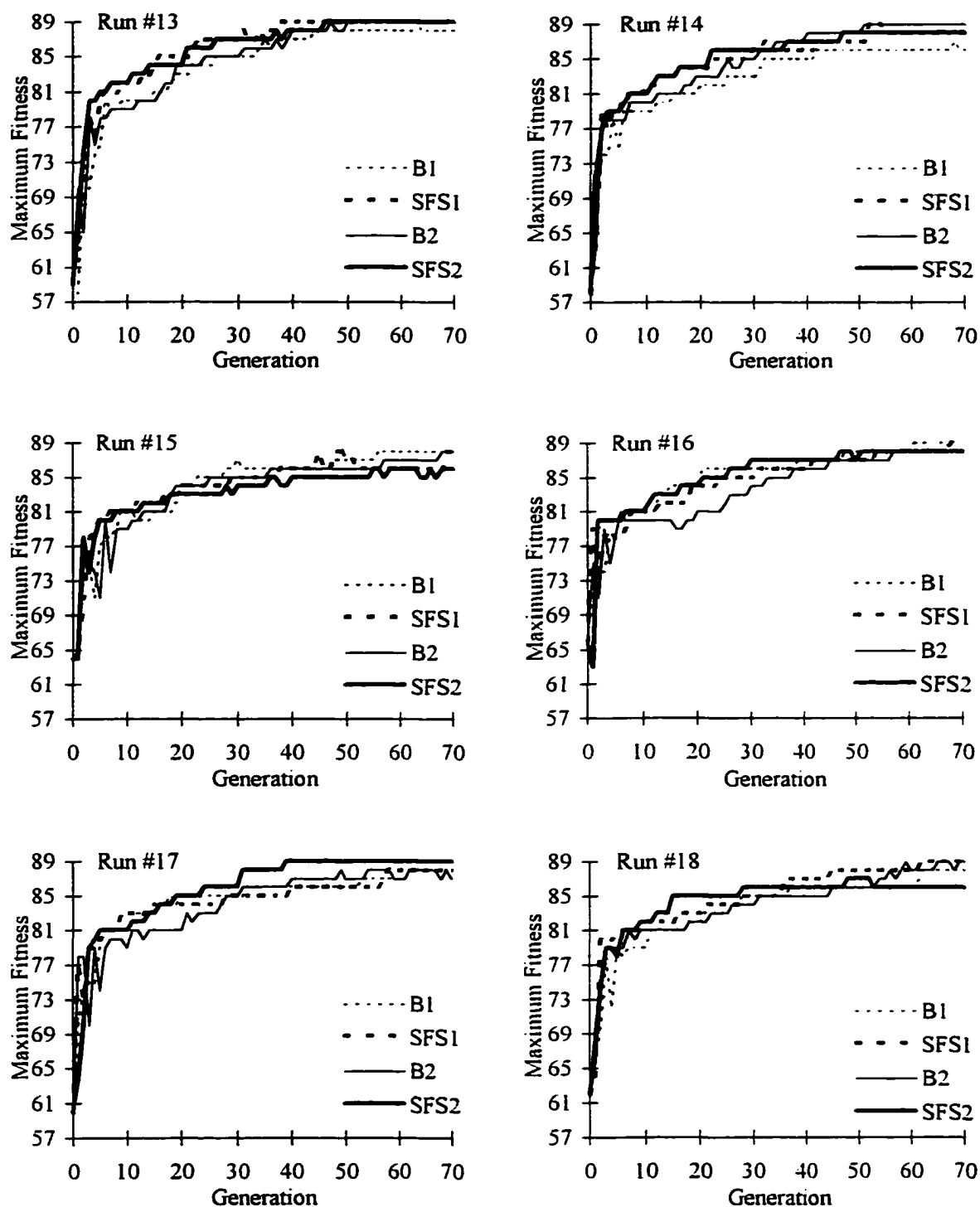


Figure C.3a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

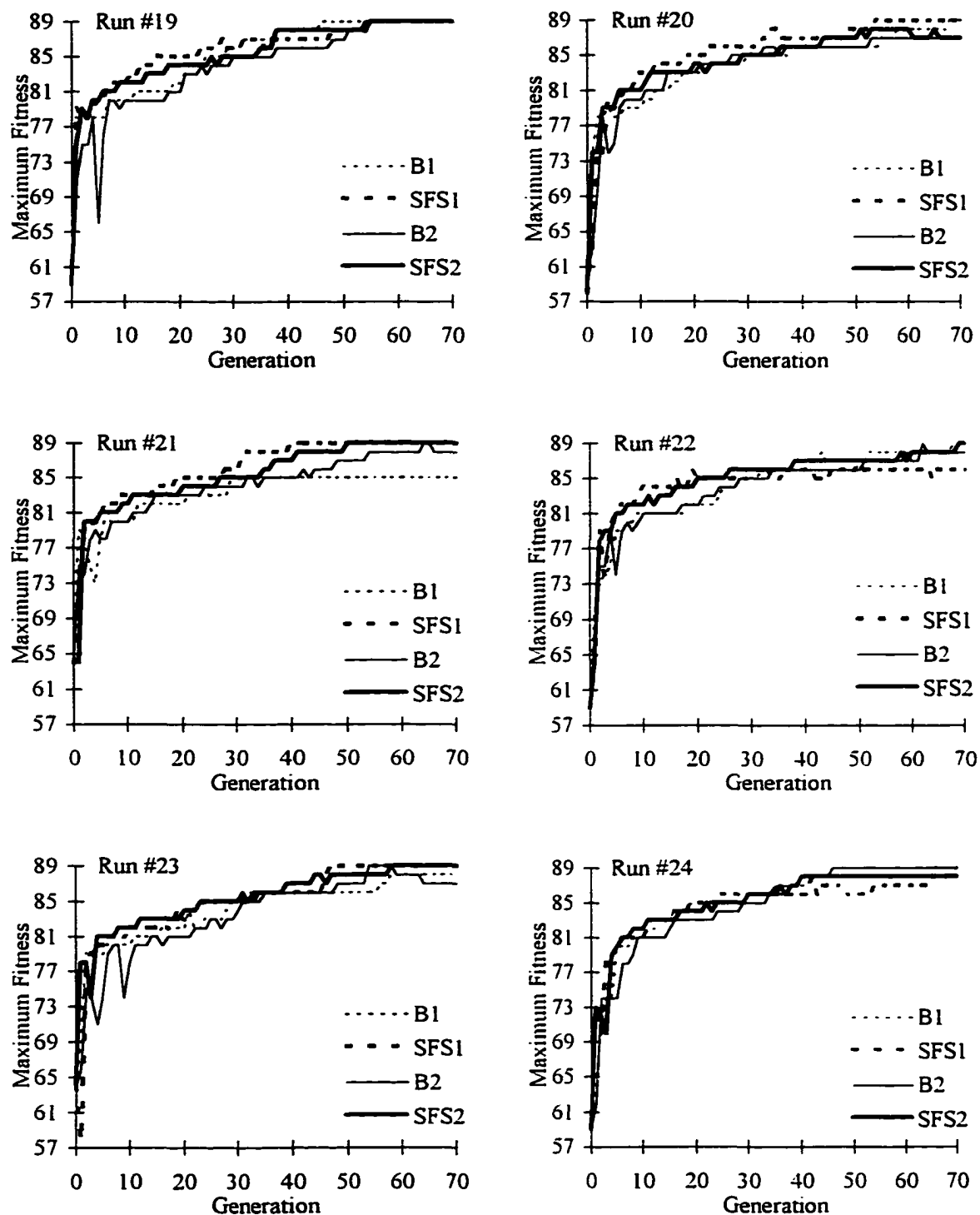


Figure C.3a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B_2 and SFS_2) and without competition (B_1 and SFS_1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

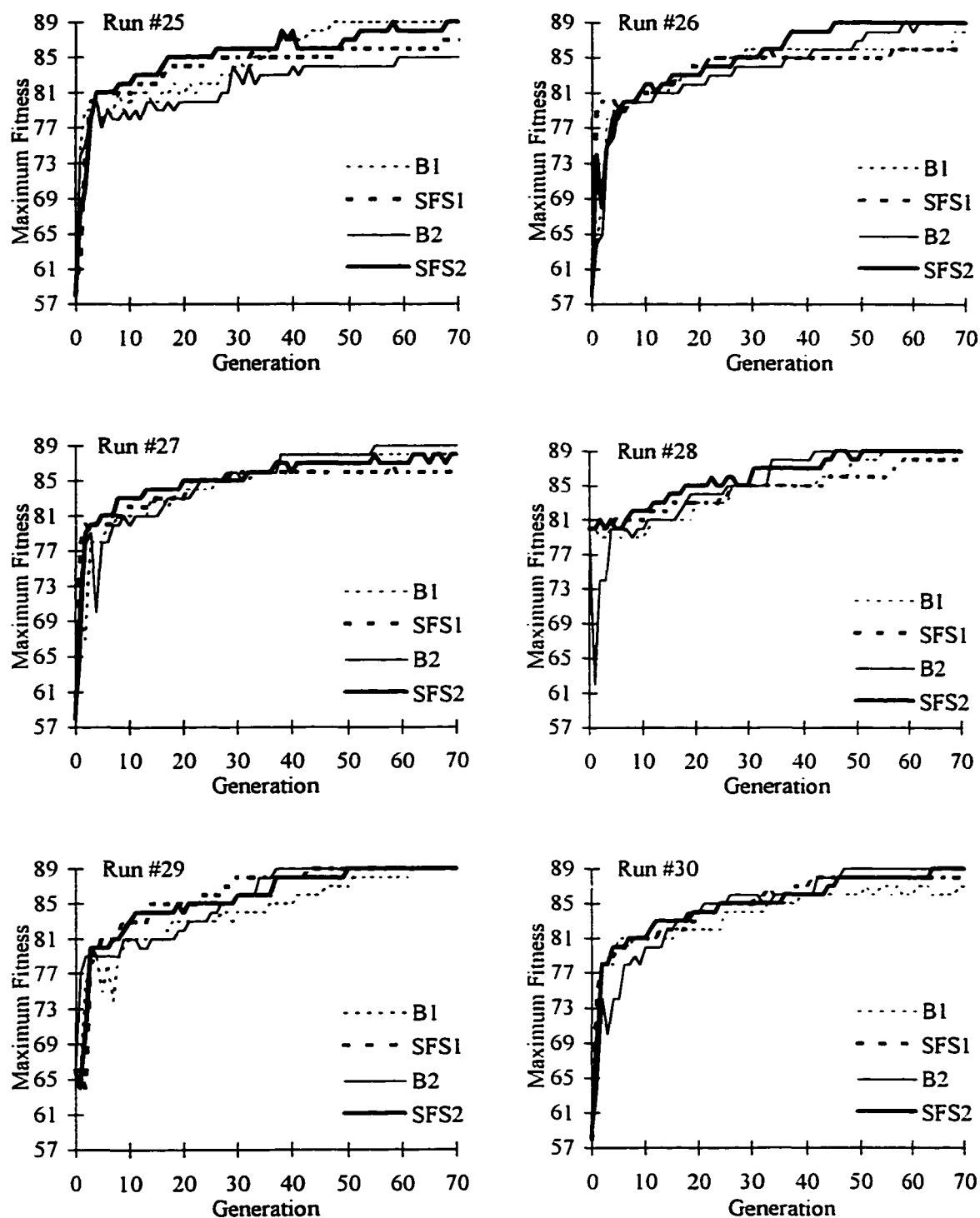


Figure C.3a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

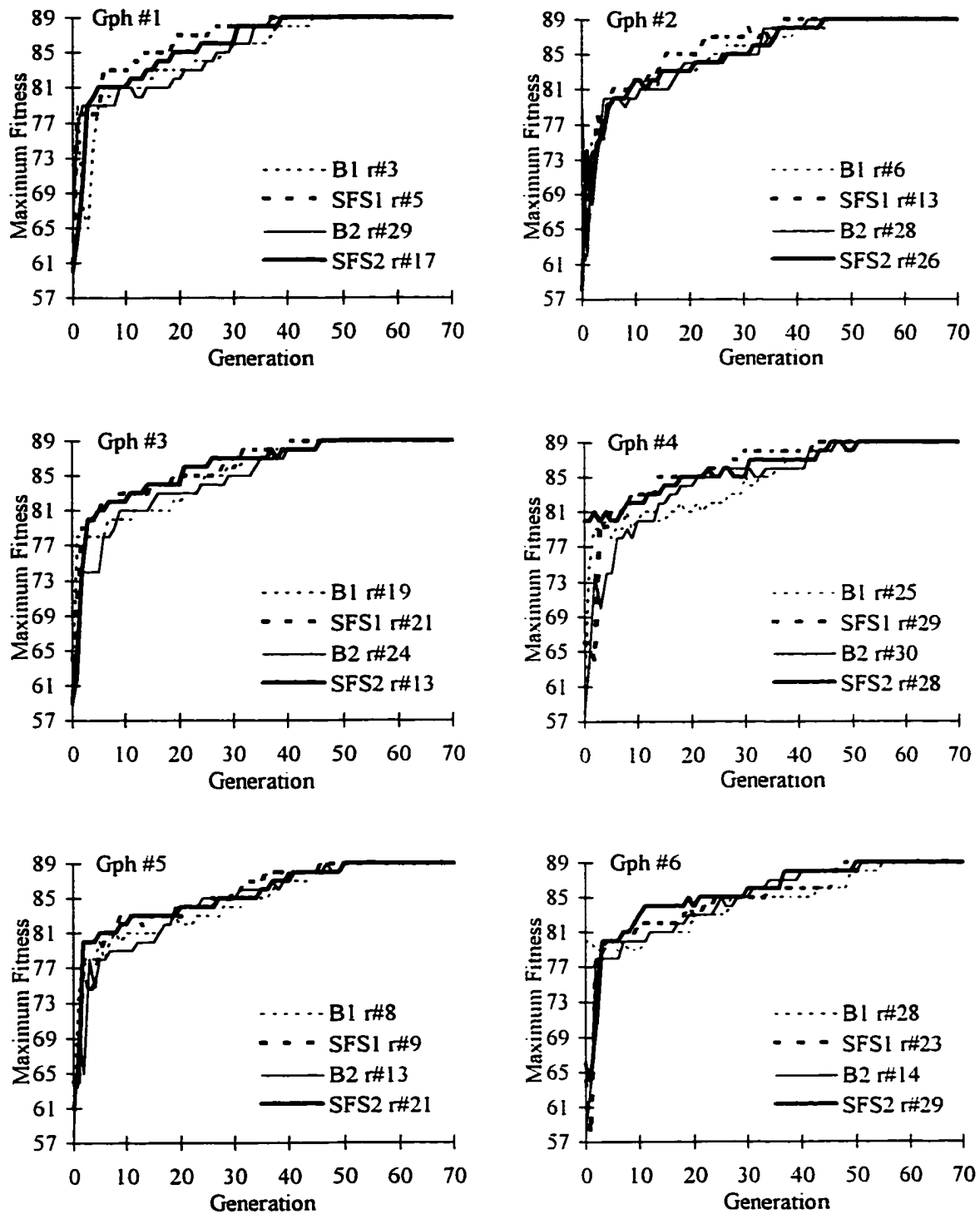


Figure C.3b: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

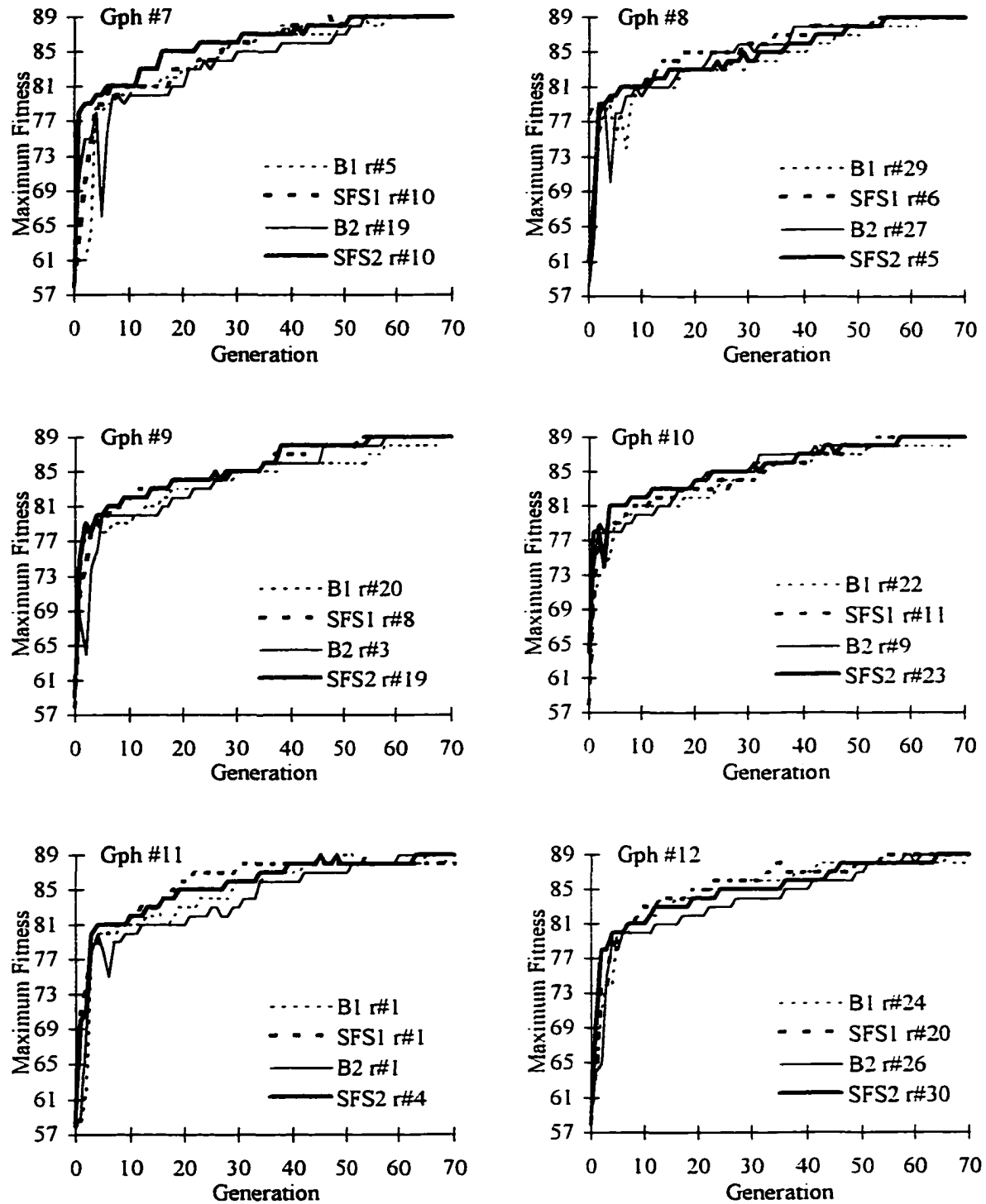


Figure C.3b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

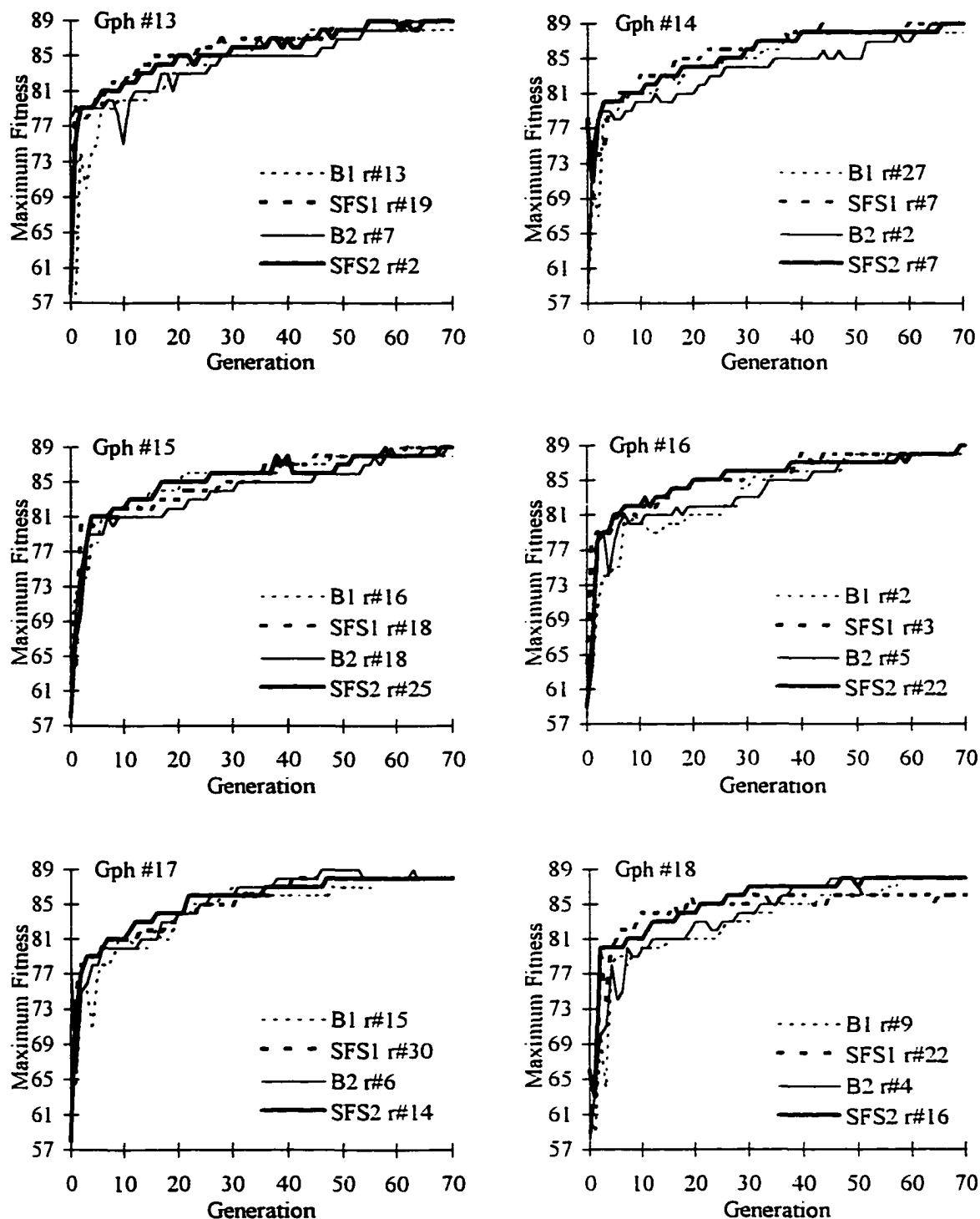


Figure C.3b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

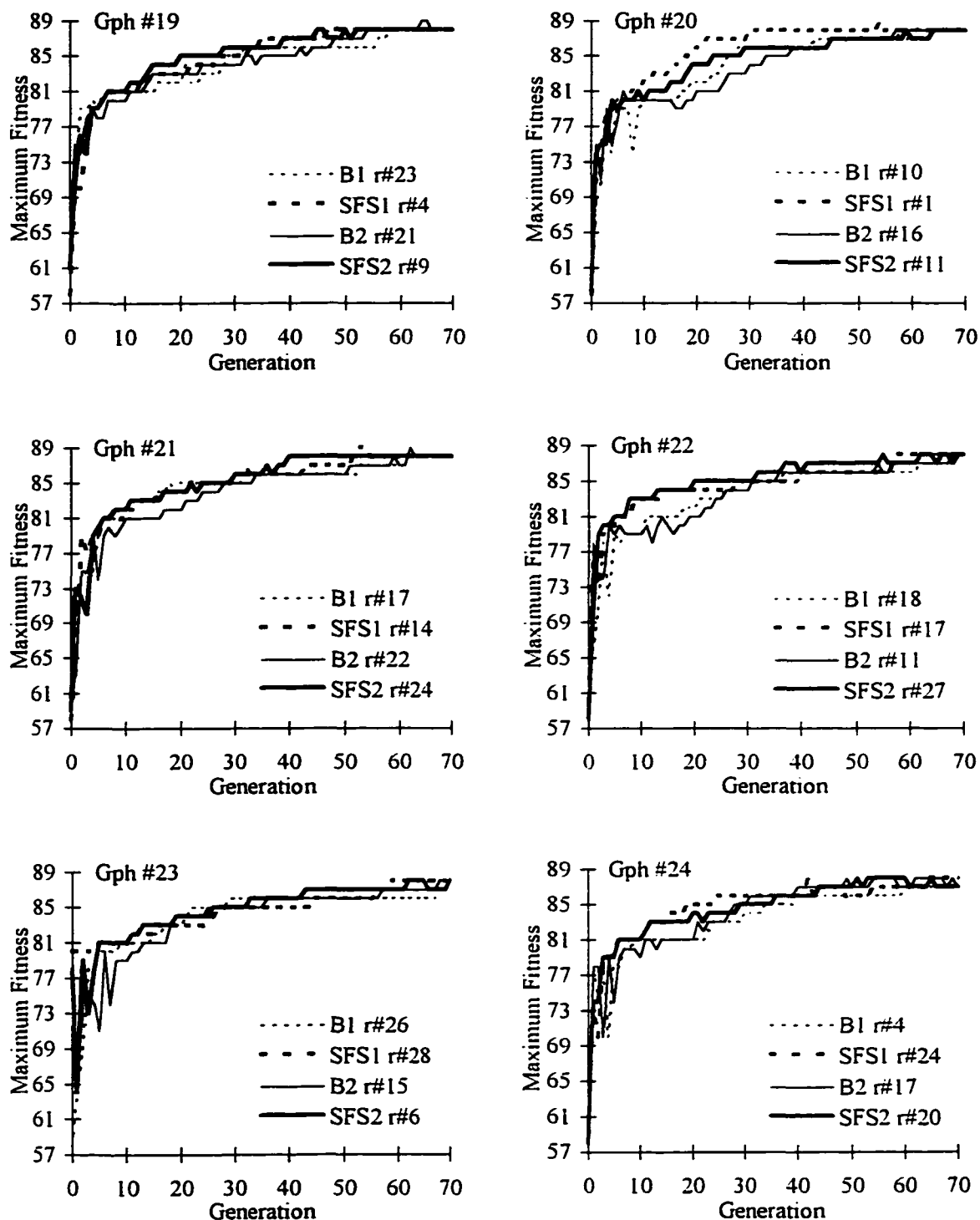


Figure C.3b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

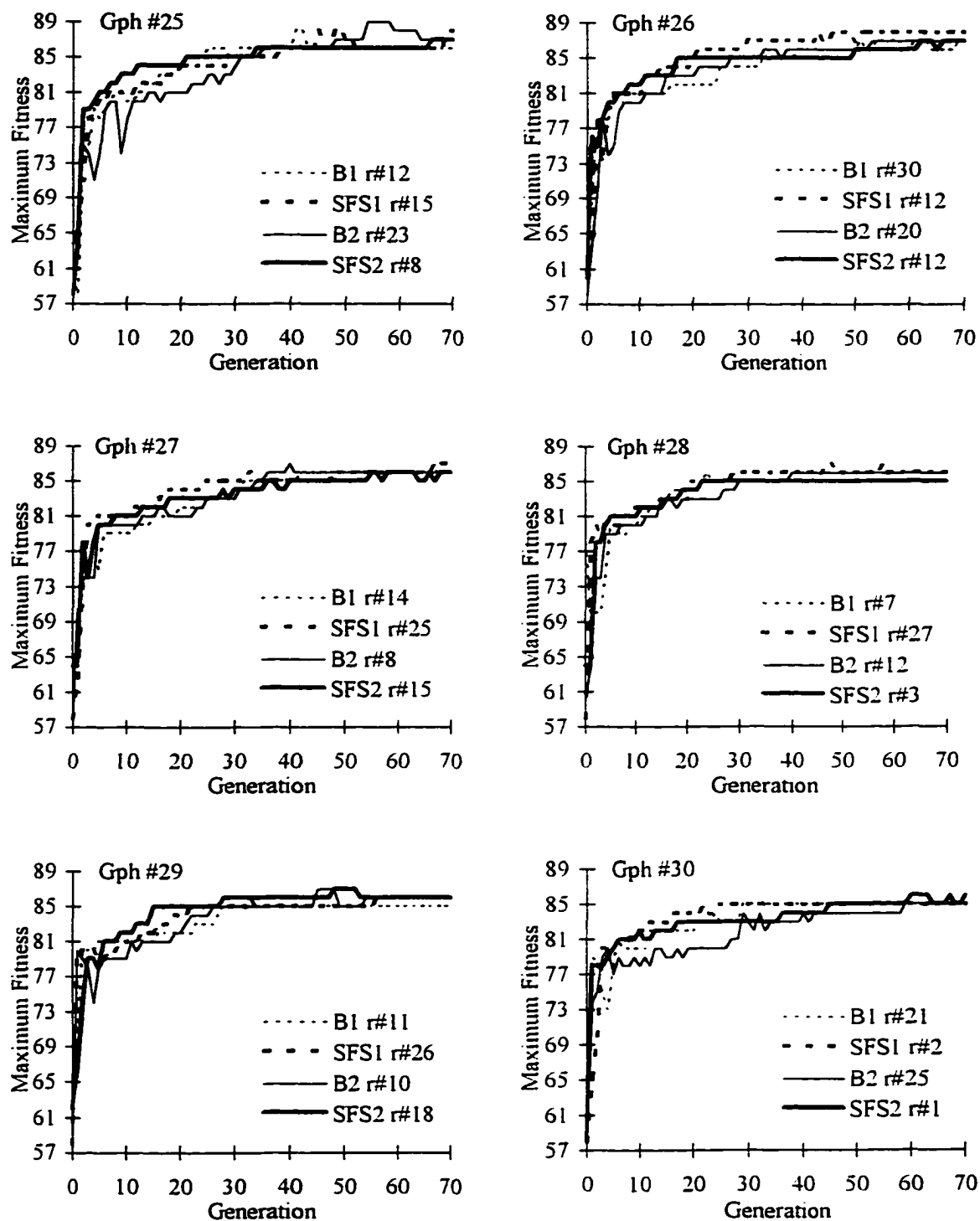


Figure C.3b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

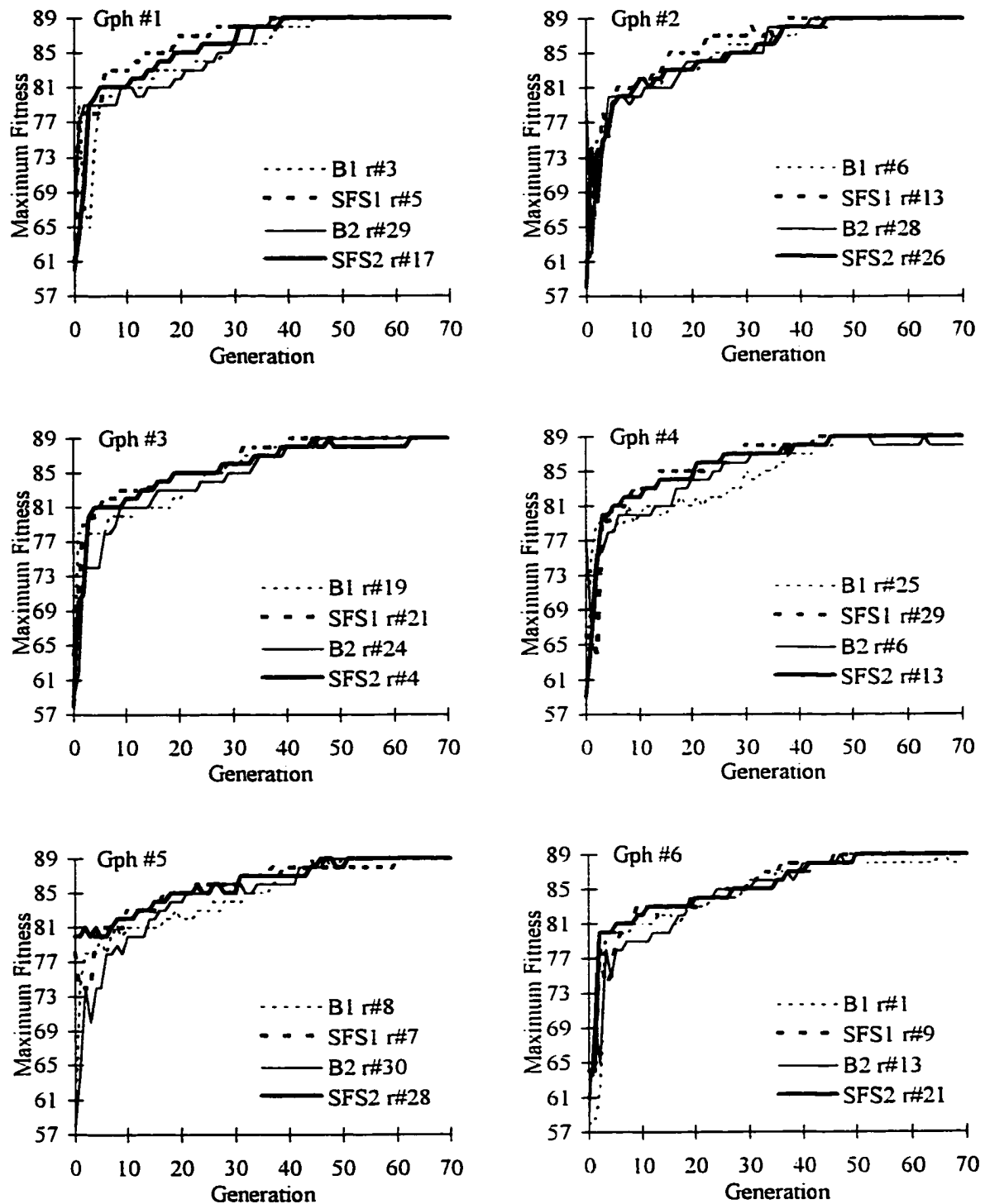


Figure C.3c: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

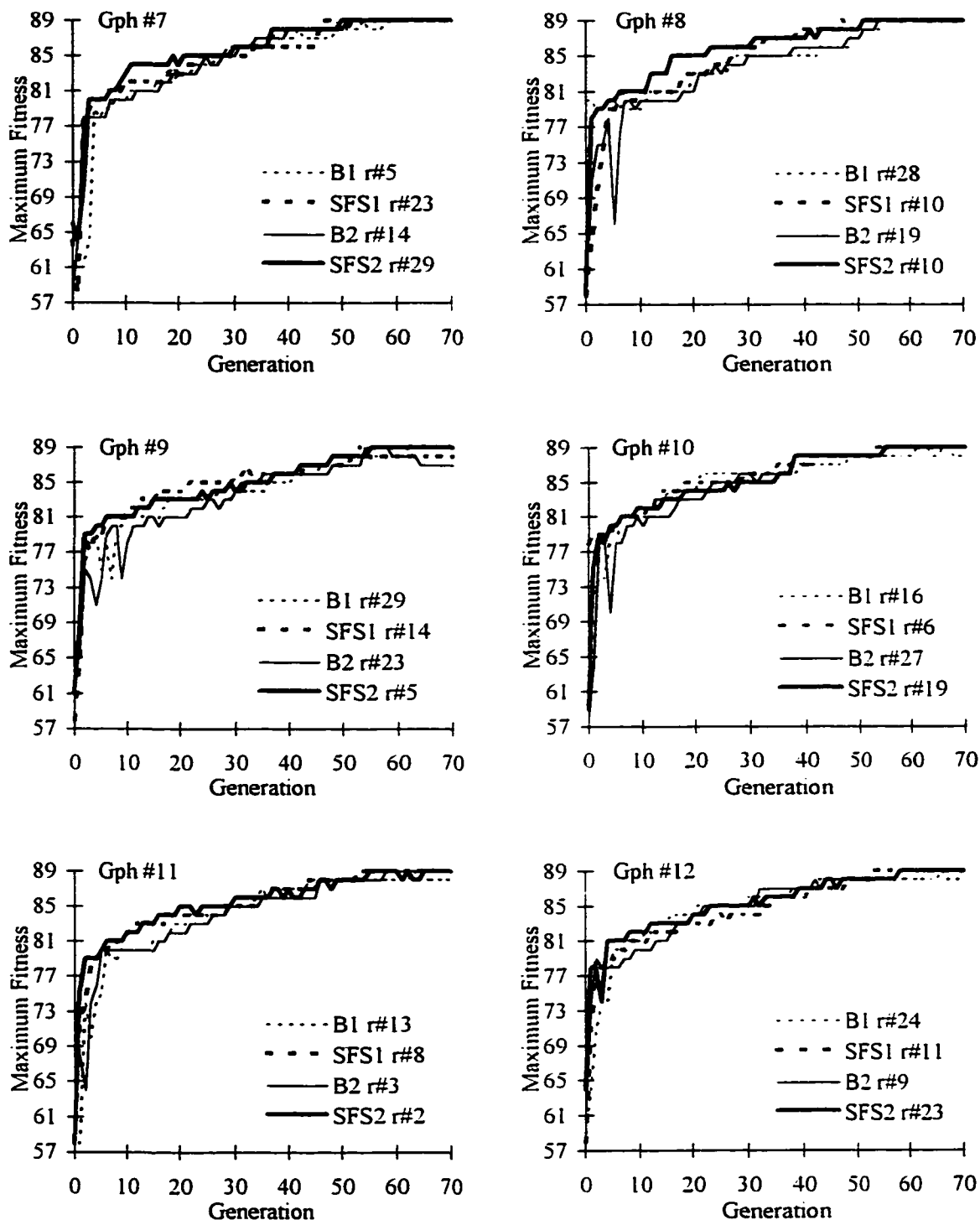


Figure C.3c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

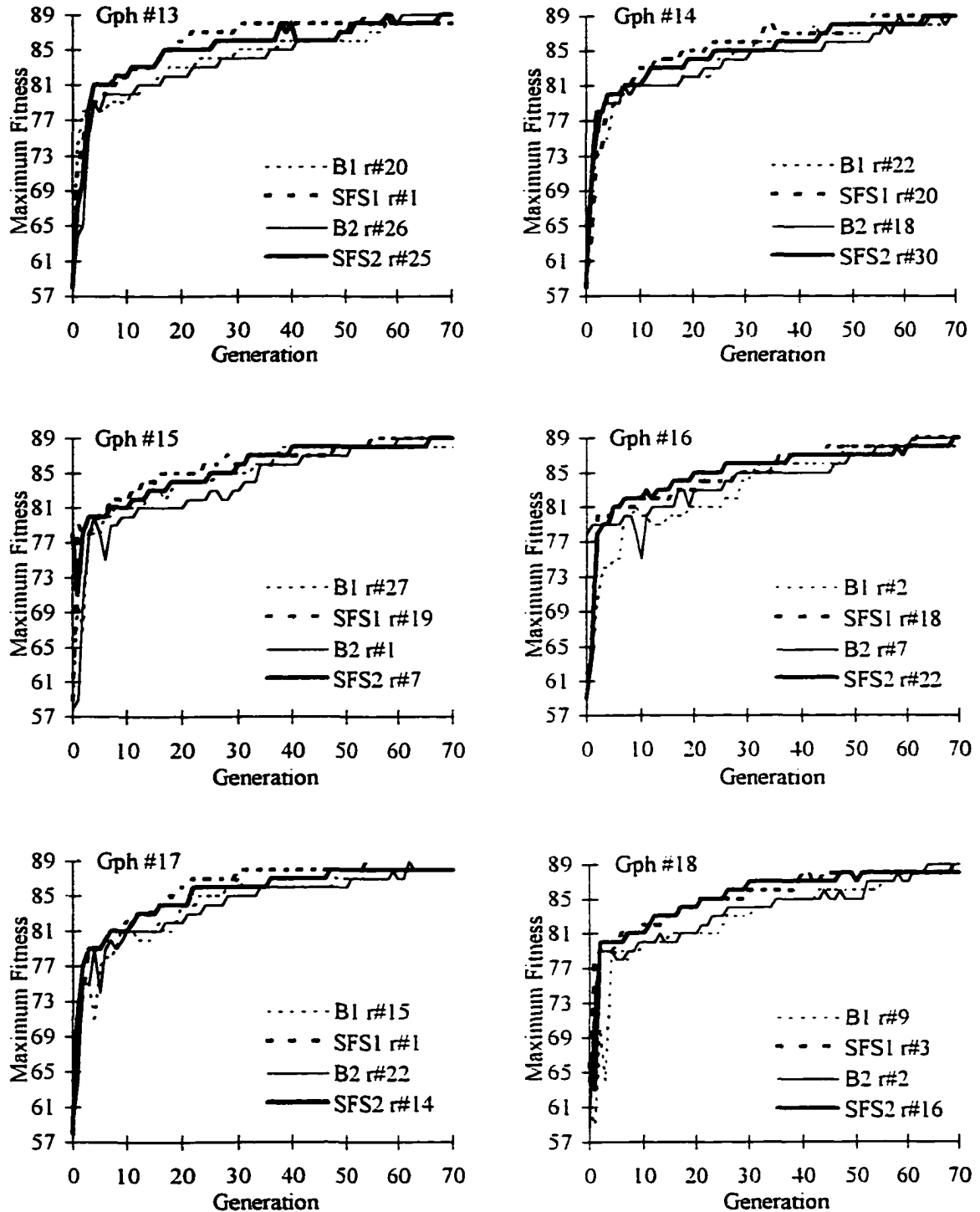


Figure C.3c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

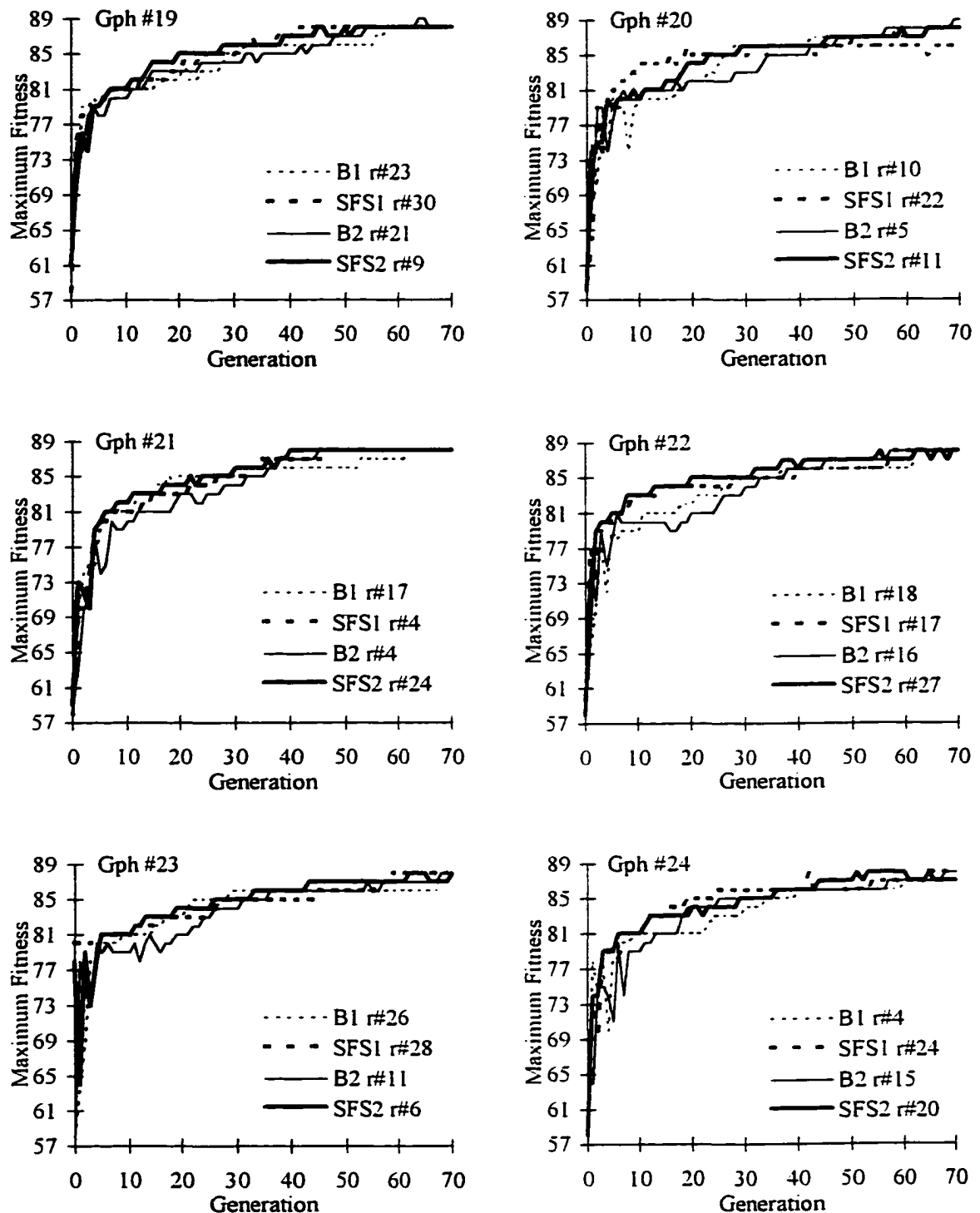


Figure C.3c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

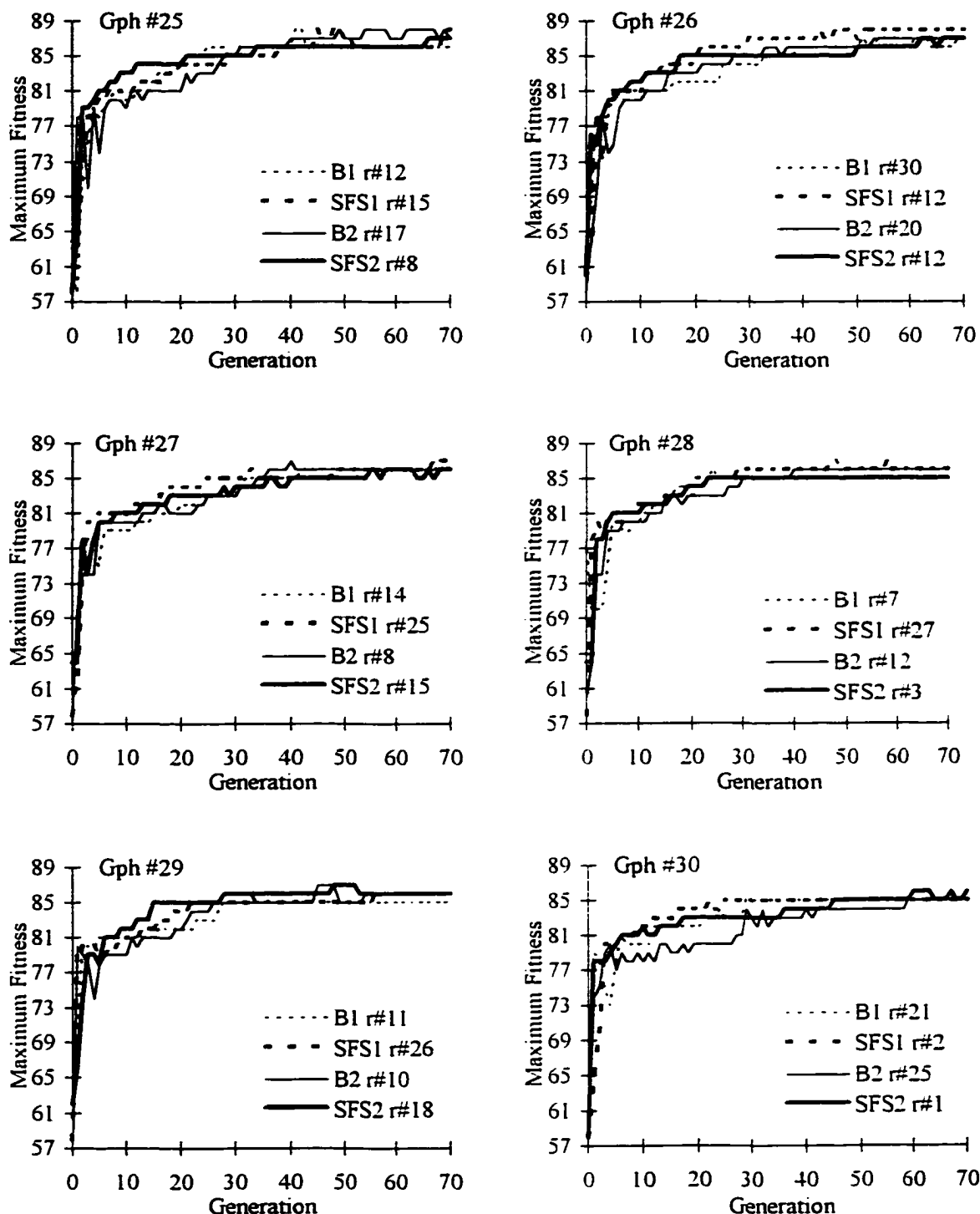


Figure C.3c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

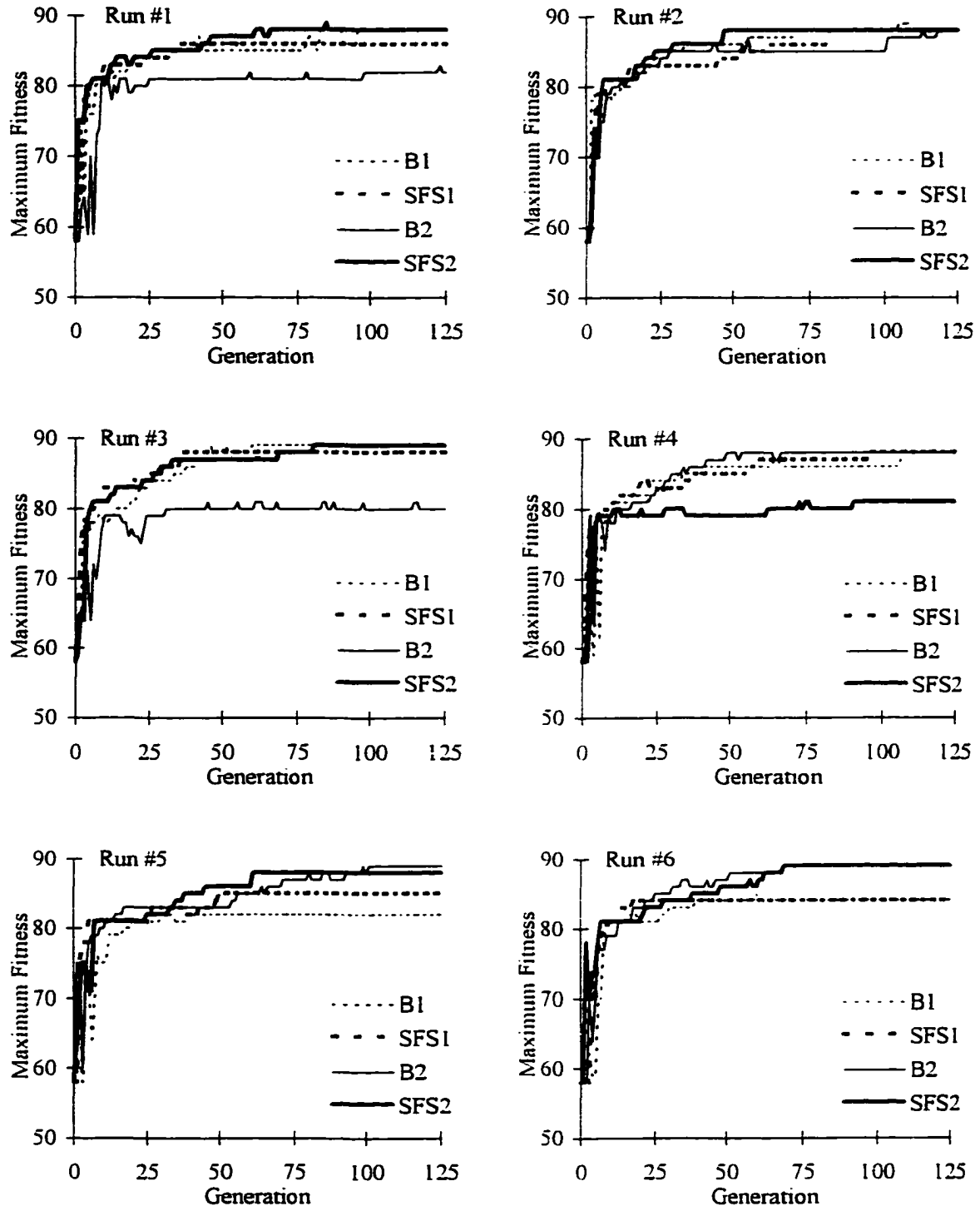


Figure C.4a: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

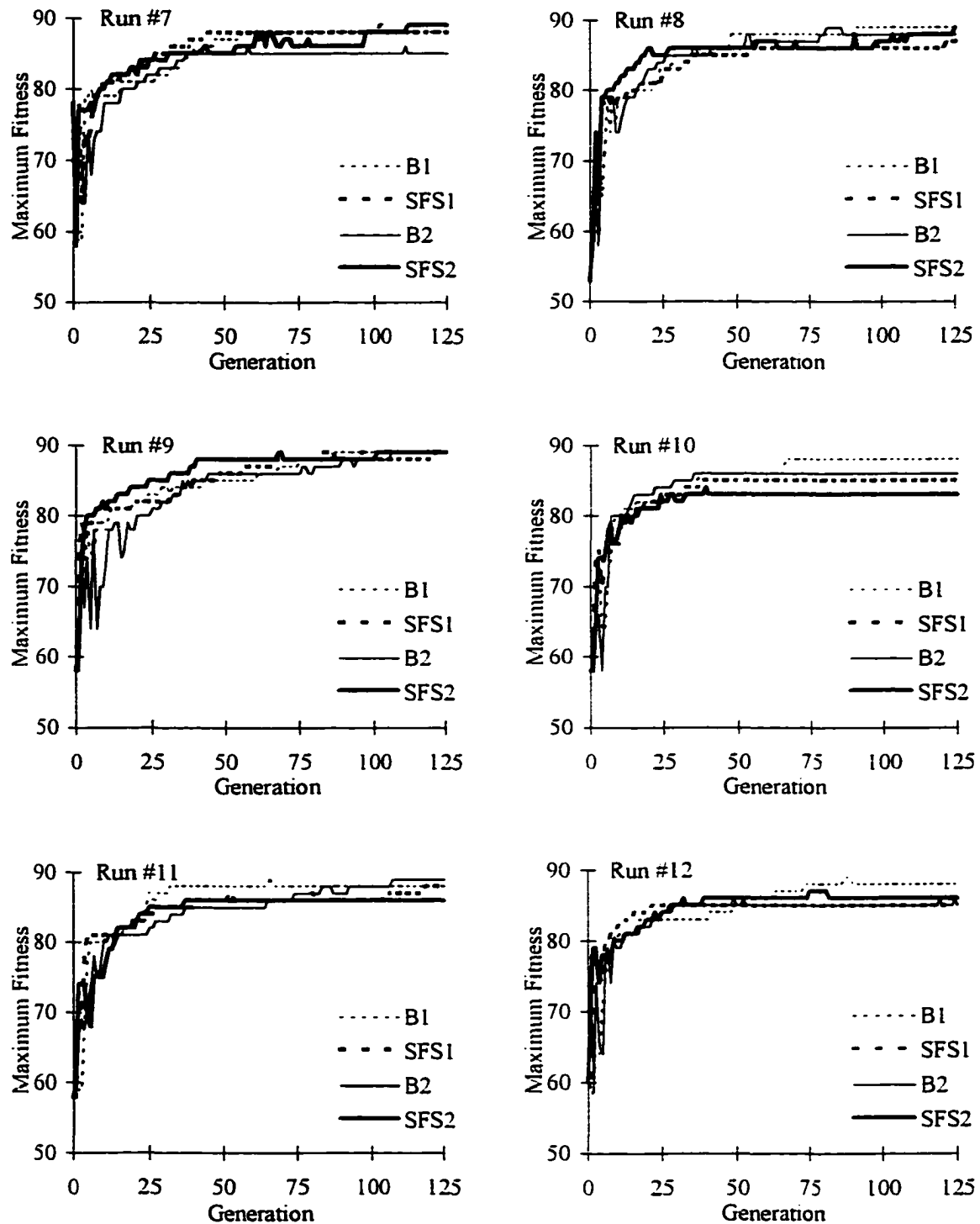


Figure C.4a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

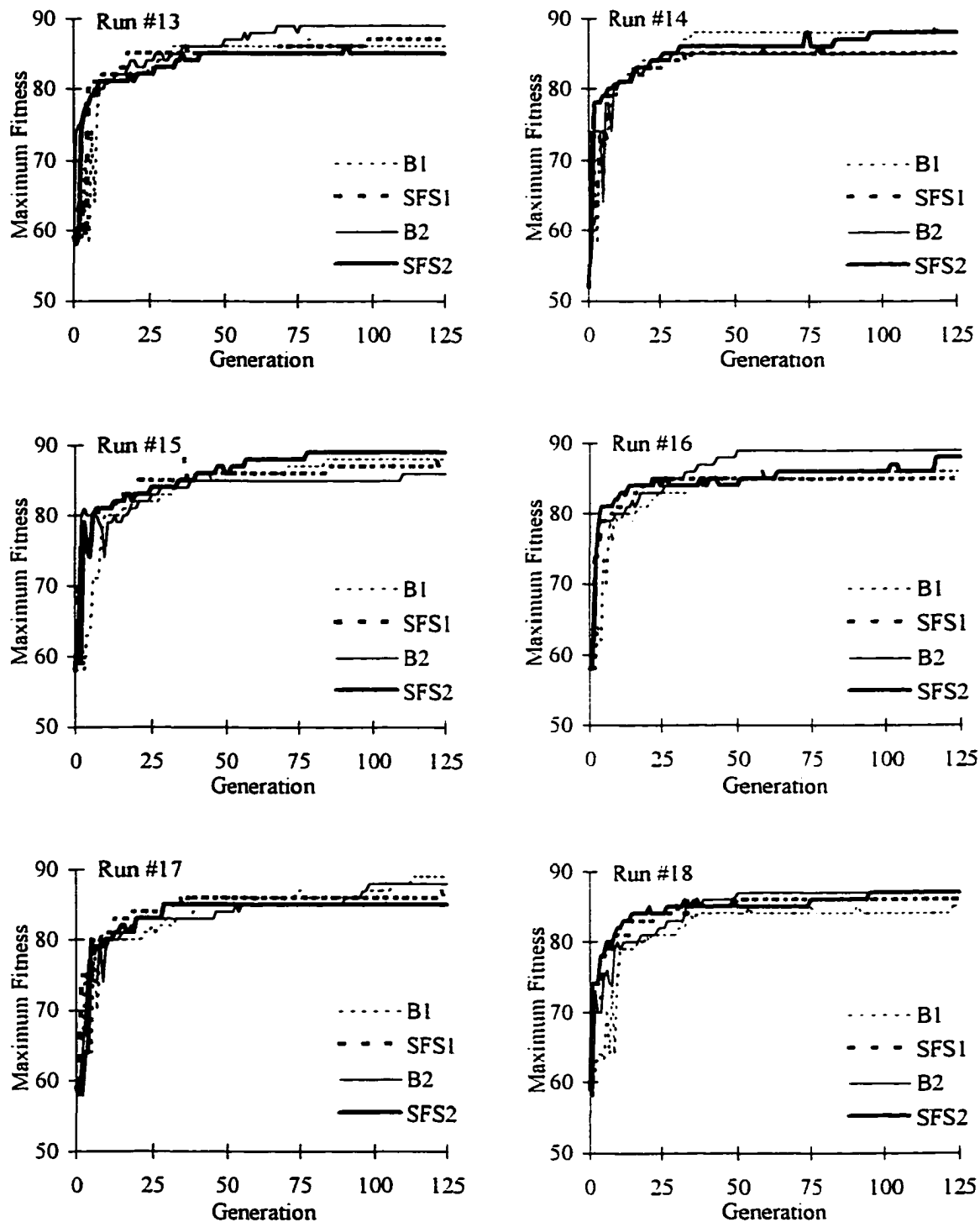


Figure C.4a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

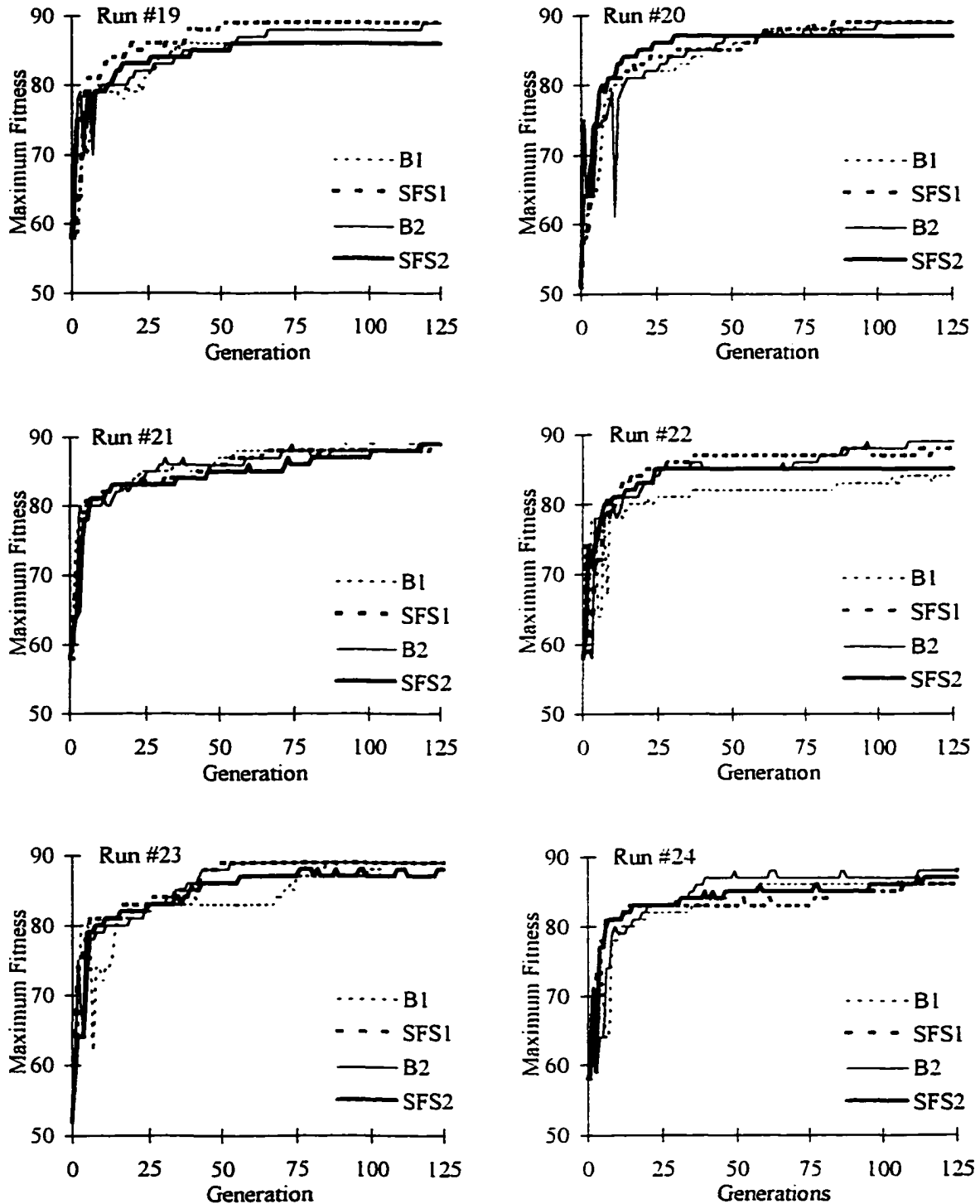


Figure C.4a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

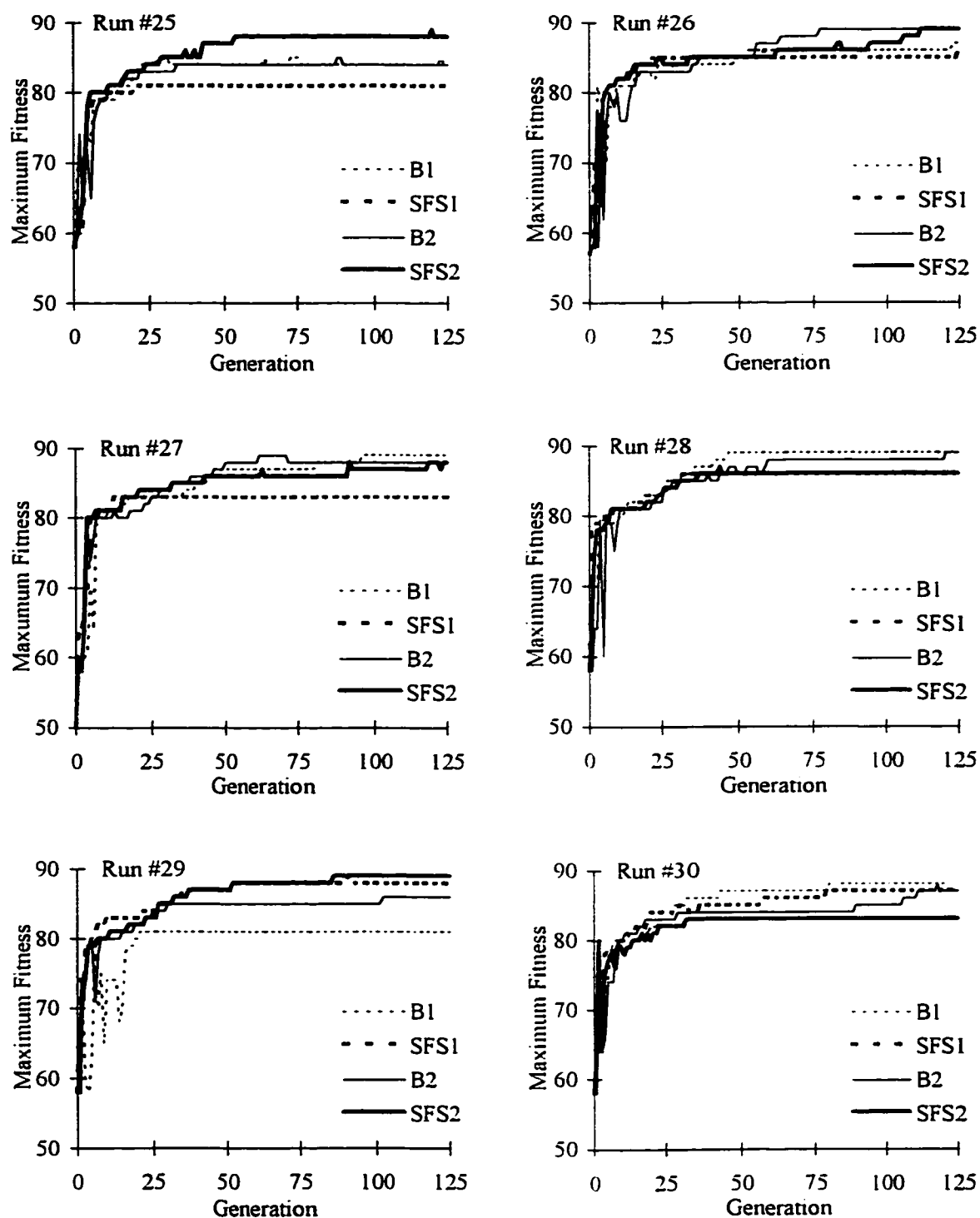


Figure C.4a continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B_2 and SFS_2) and without competition (B_1 and SFS_1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

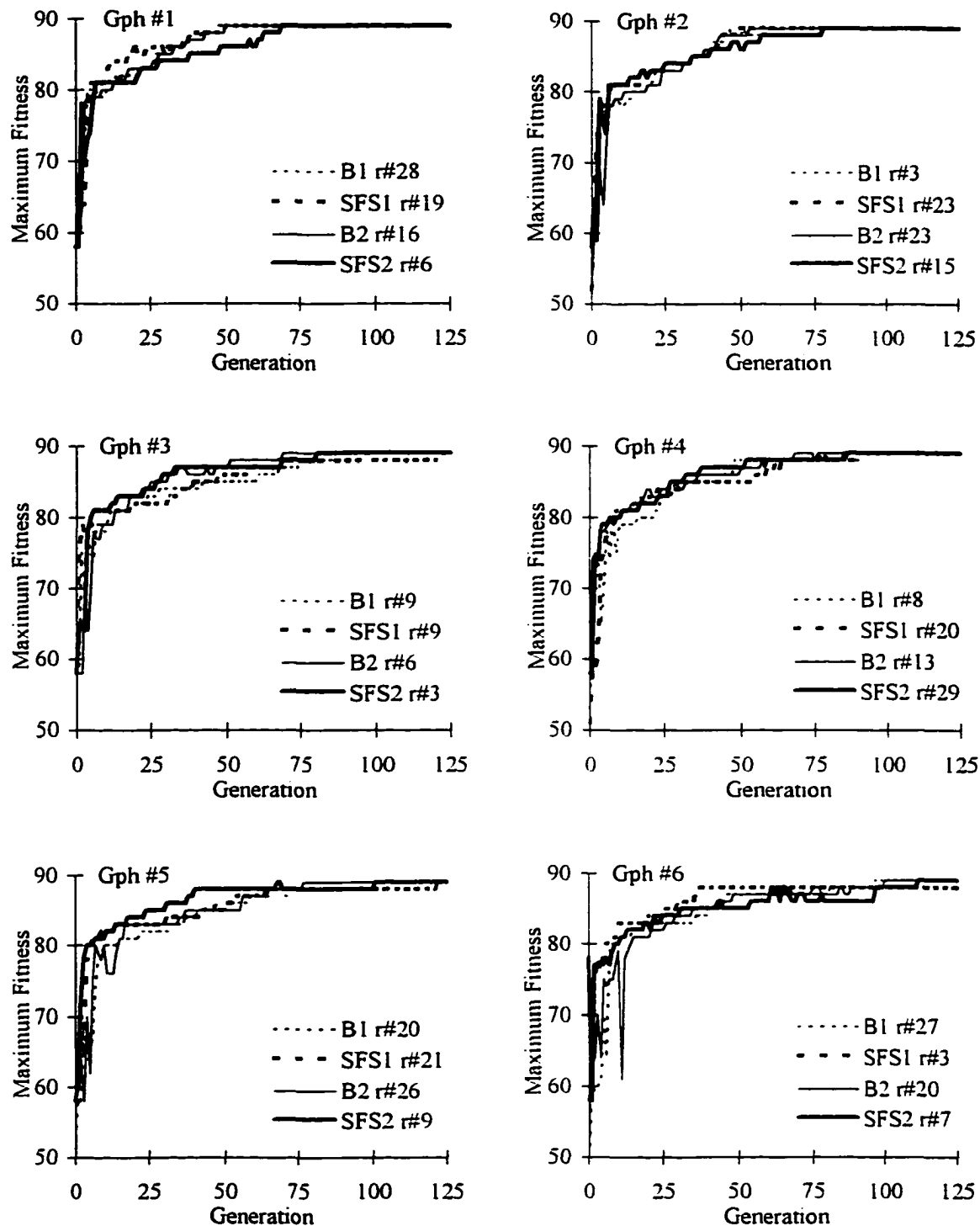


Figure C.4b: Comparison of benchmark (Bi) and benchmark with the SFS algorithm (SFSi), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

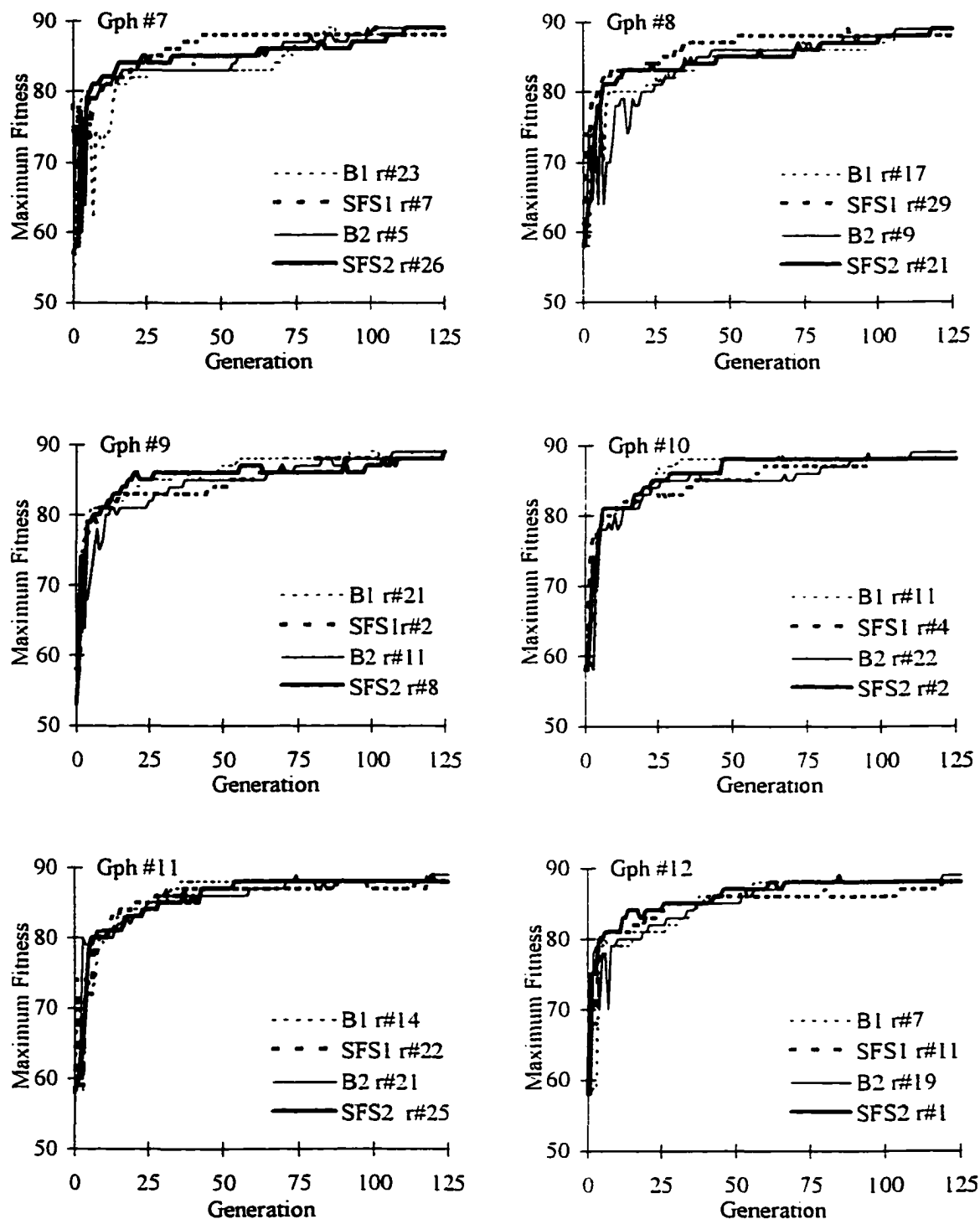


Figure C.4b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

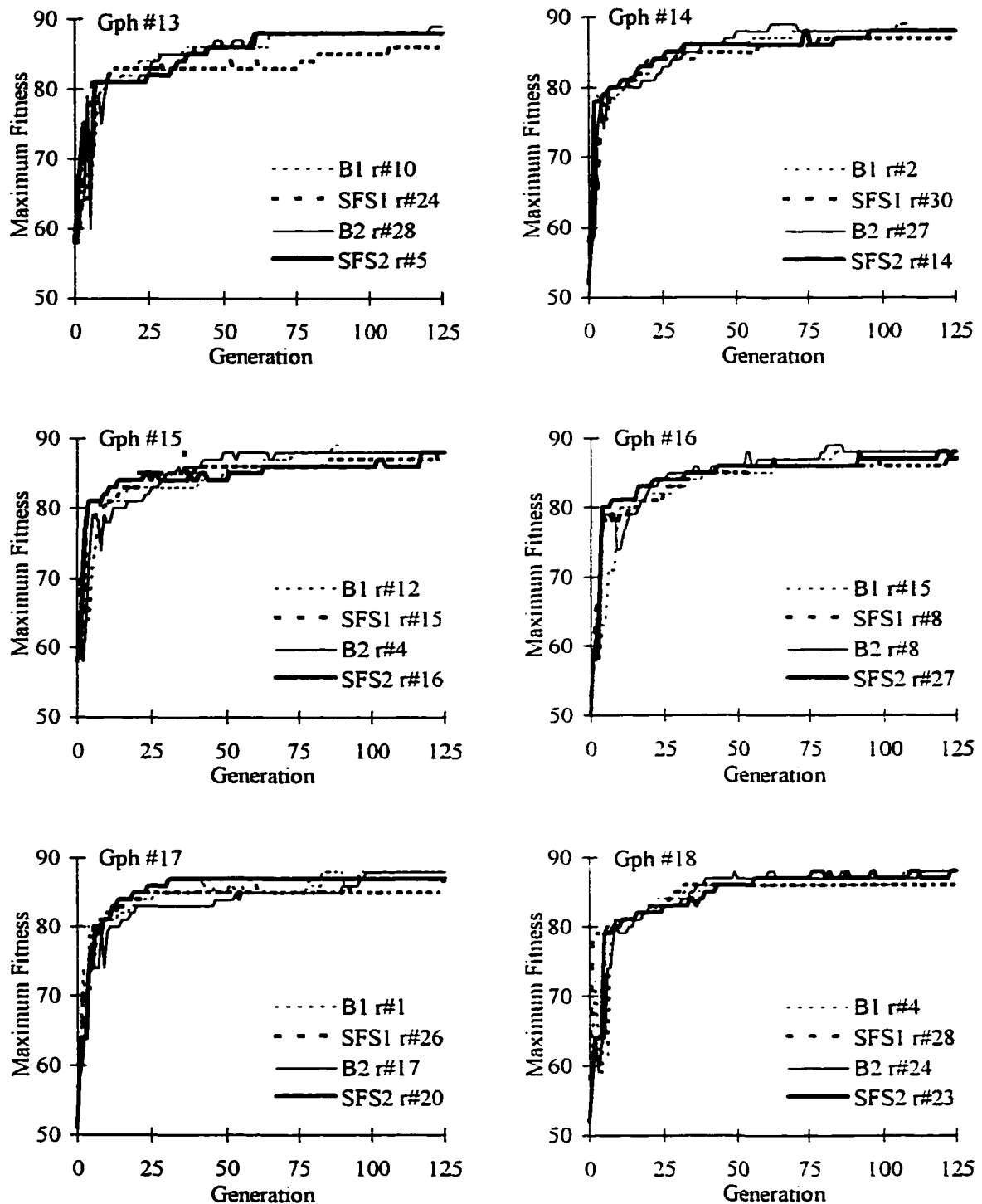


Figure C.4b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

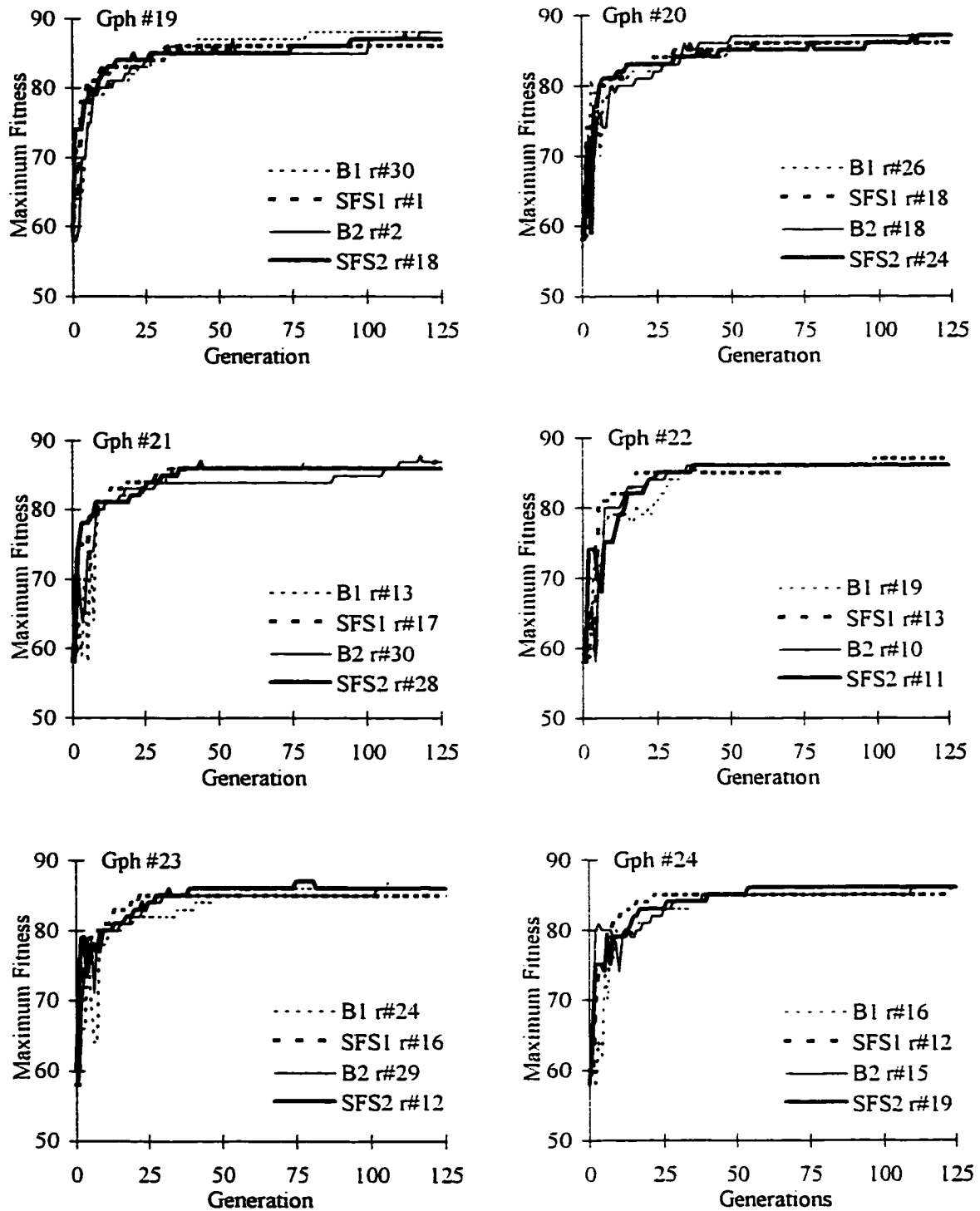


Figure C.4b continued: Comparison of benchmark (Bi) and benchmark with the SFS algorithm (SFSi), both with competition (B2 and SFS2) and without competition (B1 and SFS1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

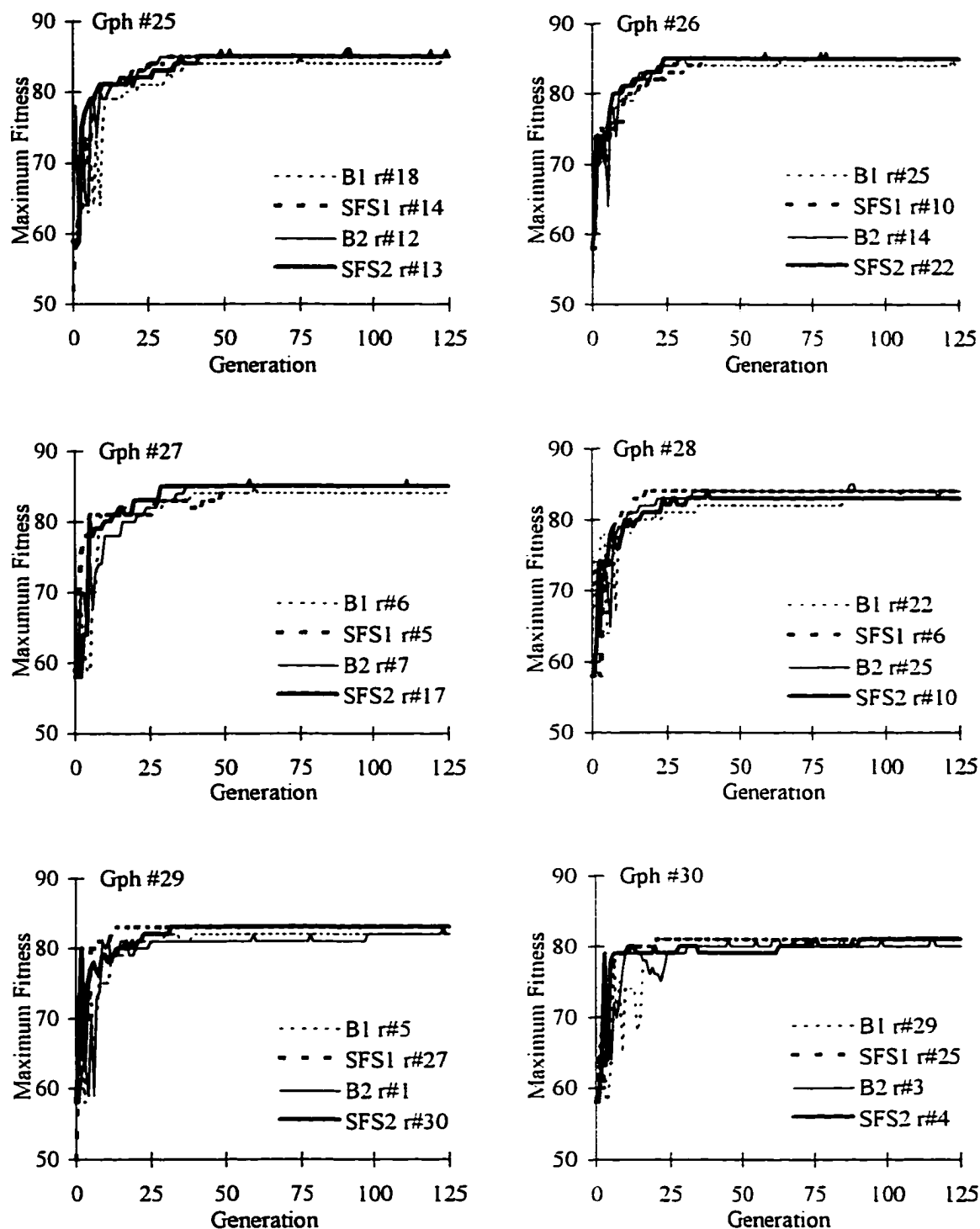


Figure C.4b continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$ and $SFS2$) and without competition ($B1$ and $SFS1$) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

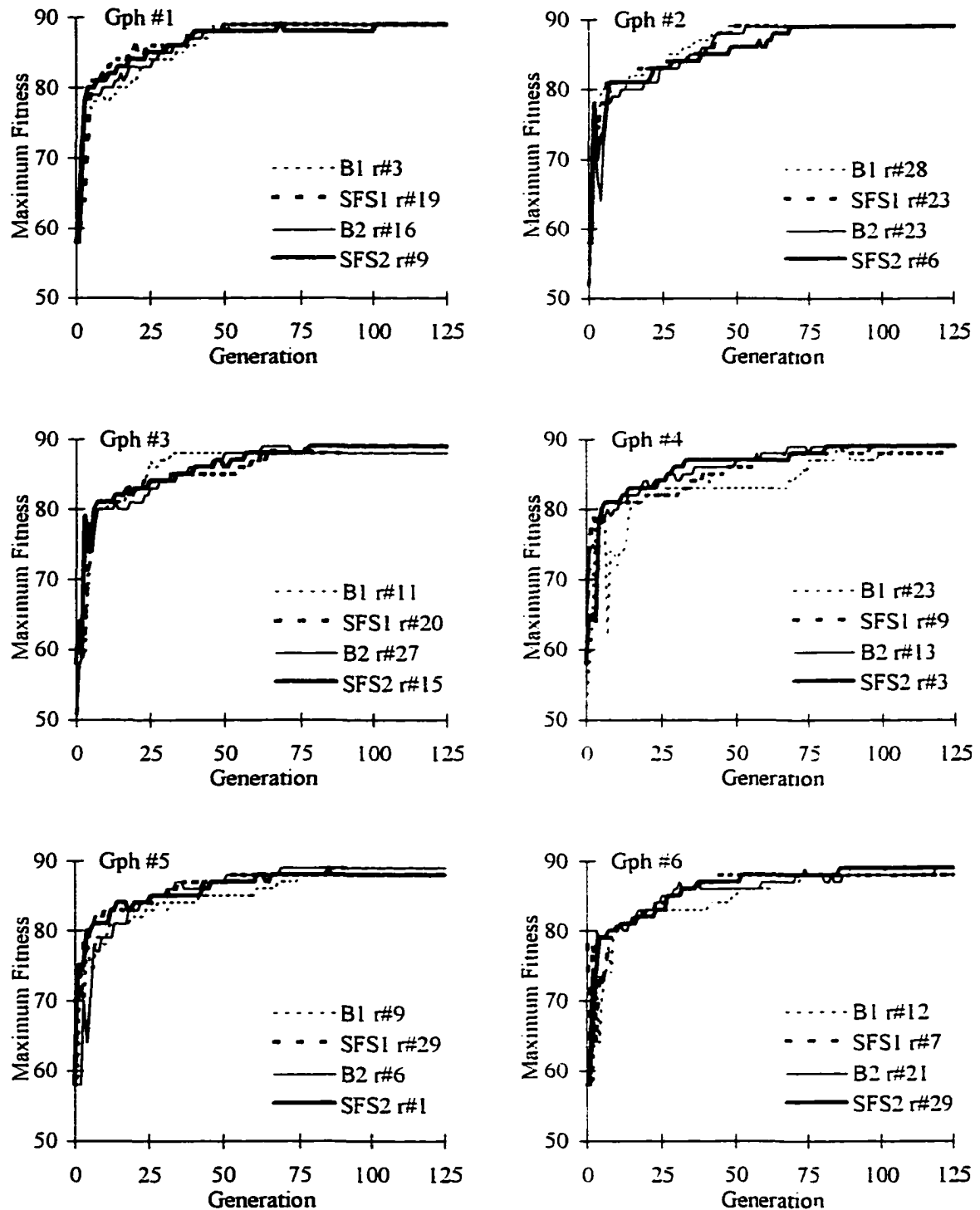


Figure C.4c: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), both with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

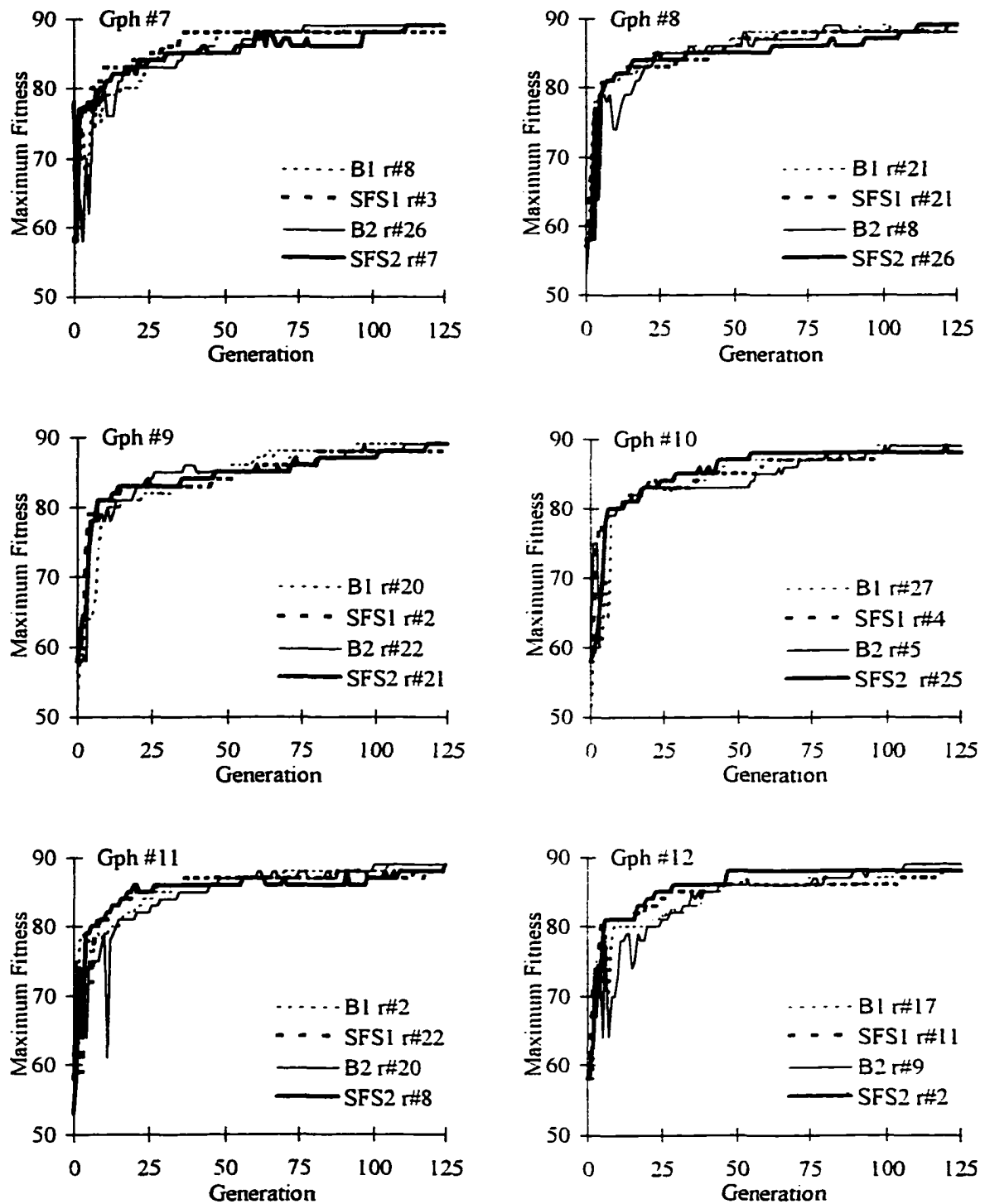


Figure C.4c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition (B2, SFS2) and without competition (B1, SFS1) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

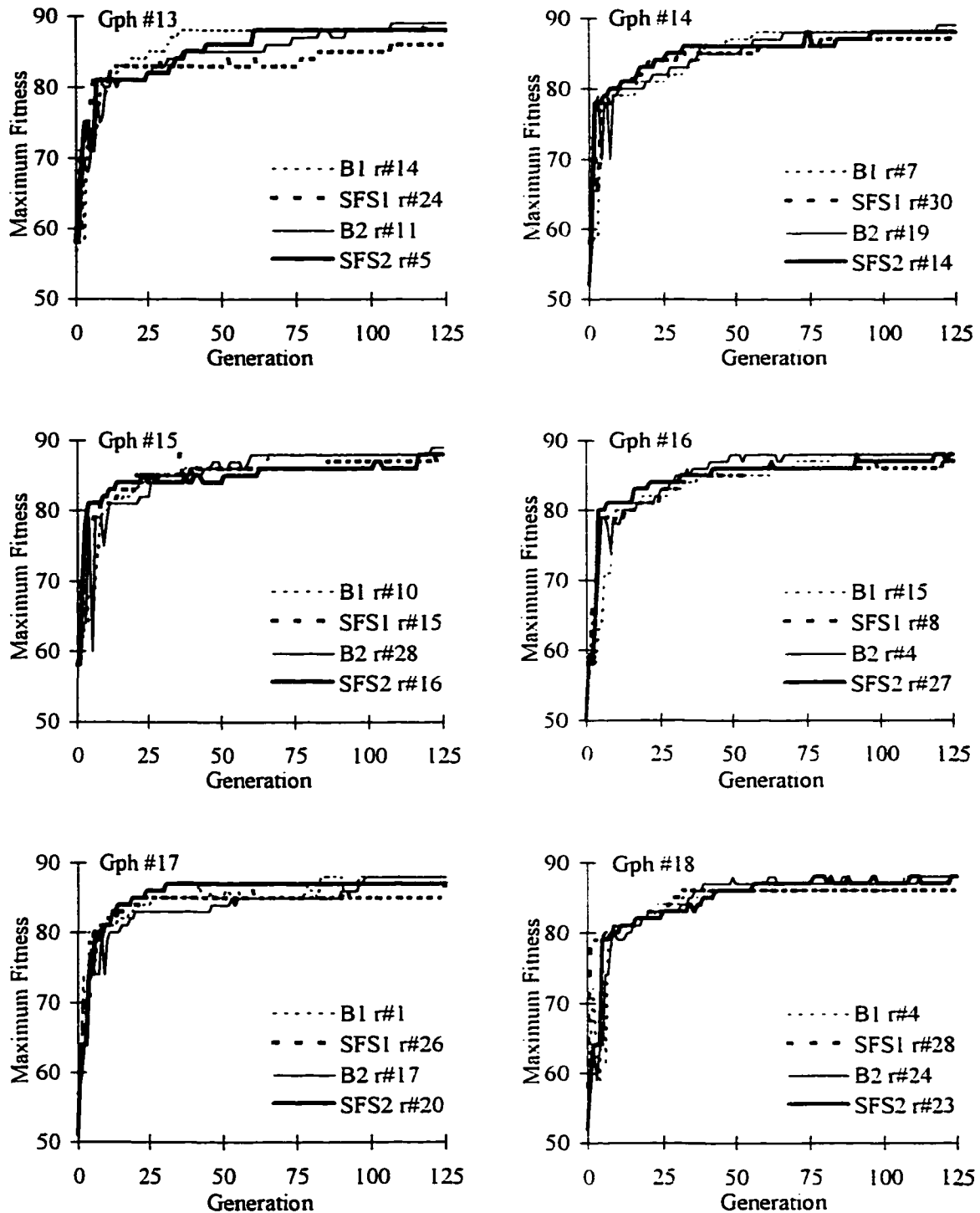


Figure C.4c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

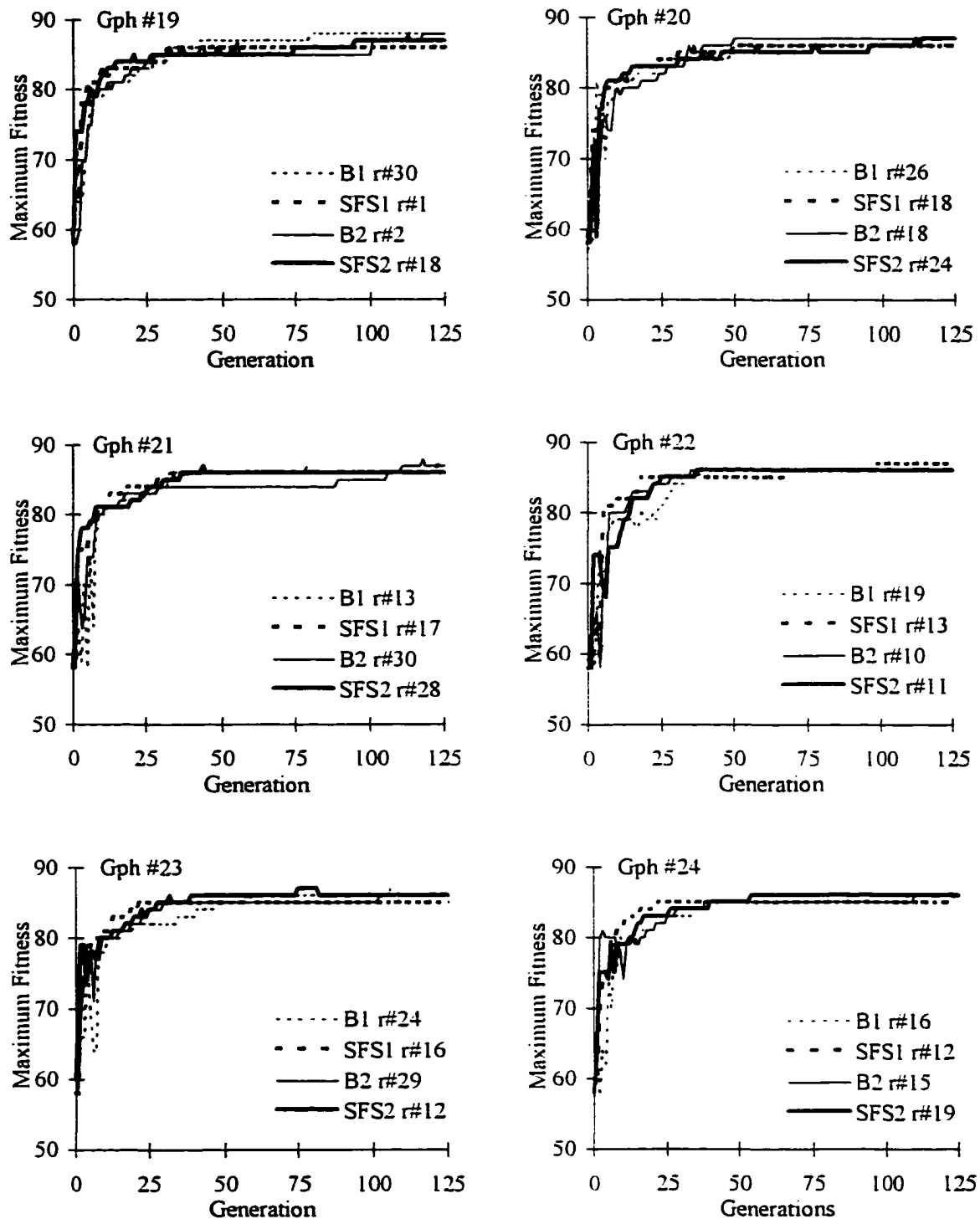


Figure C.4c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

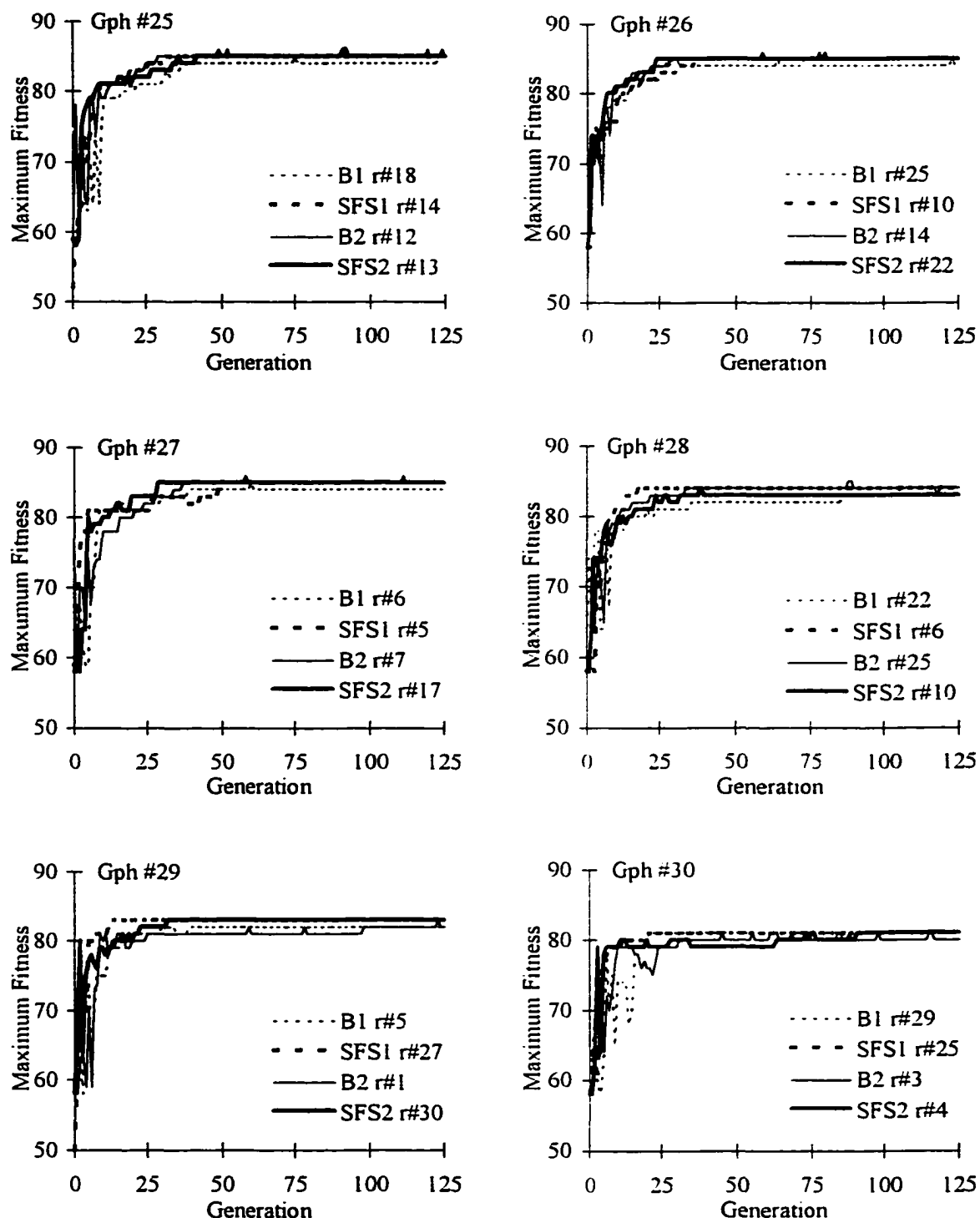


Figure C.4c continued: Comparison of benchmark (B_i) and benchmark with the SFS algorithm (SFS_i), with competition ($B2$, $SFS2$) and without competition ($B1$, $SFS1$) using a population size of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

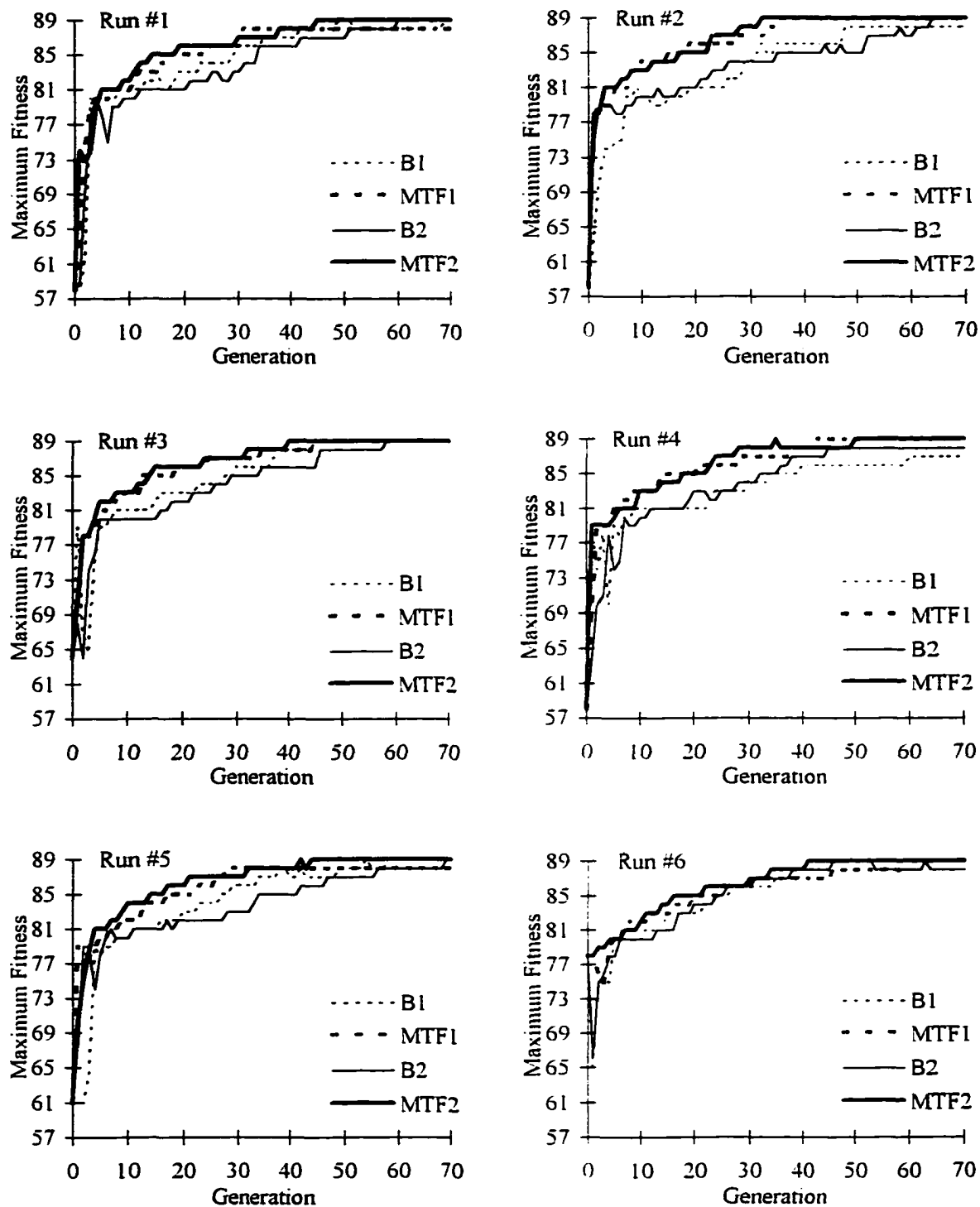


Figure C.5a: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

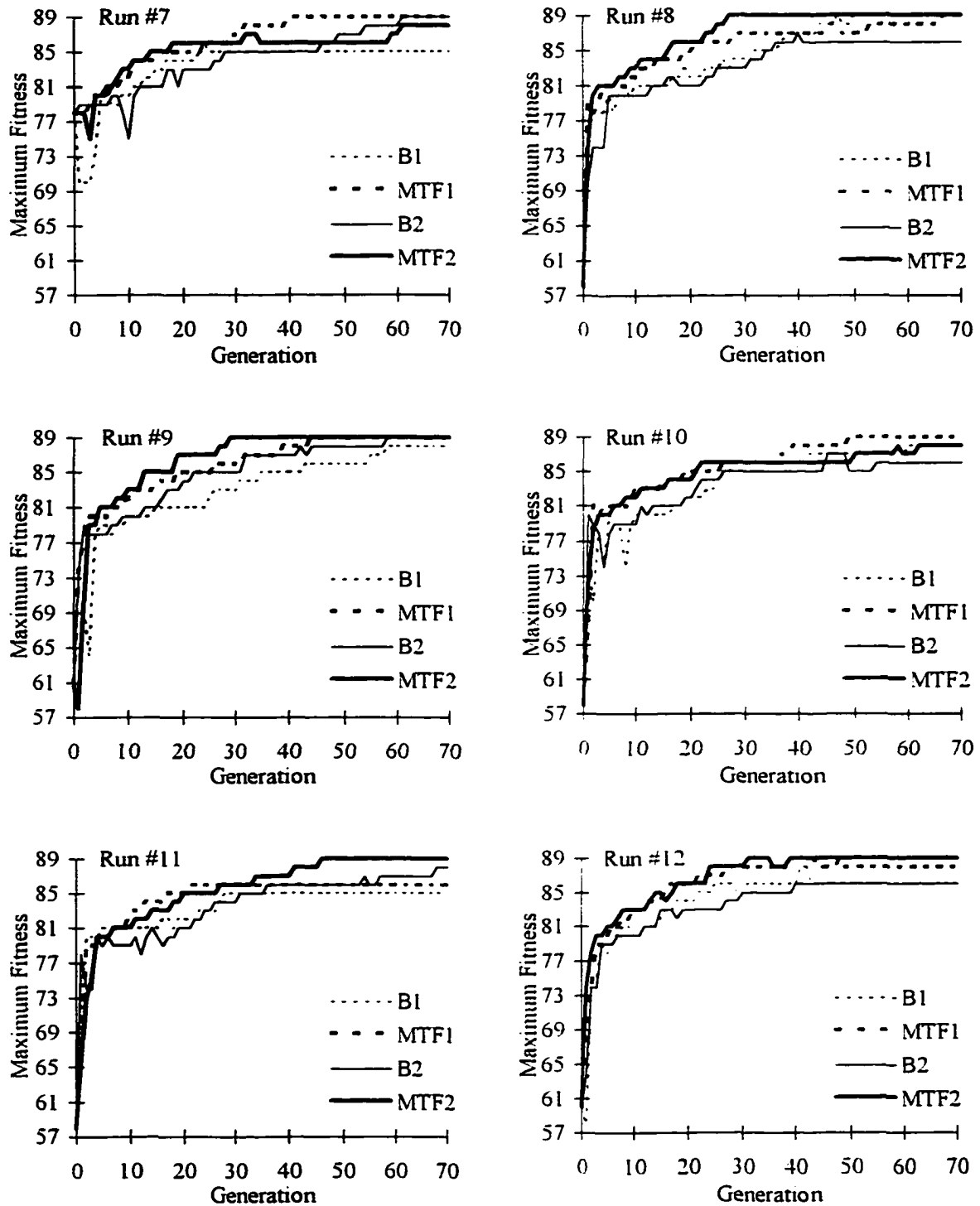


Figure C.5a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

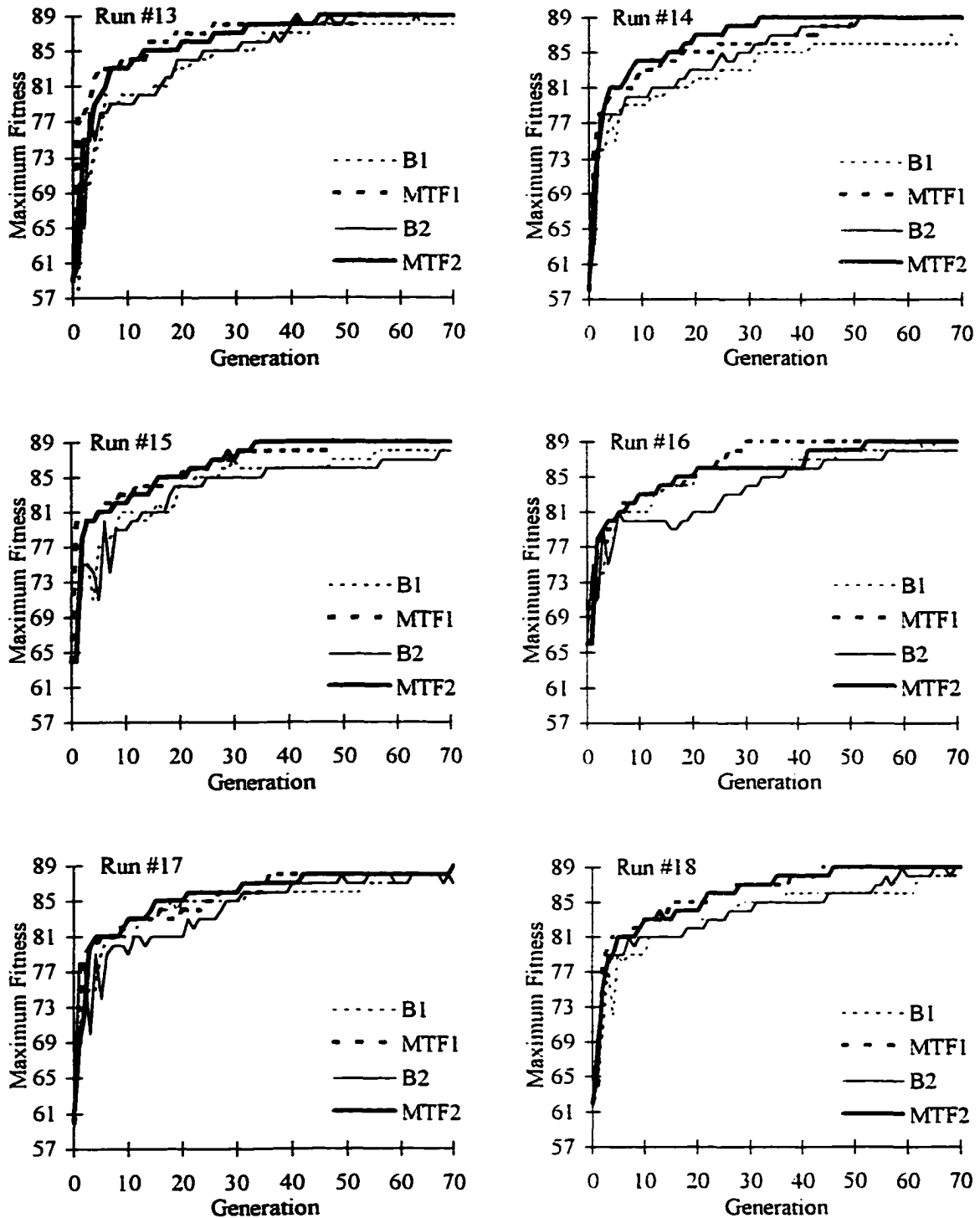


Figure C.5a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

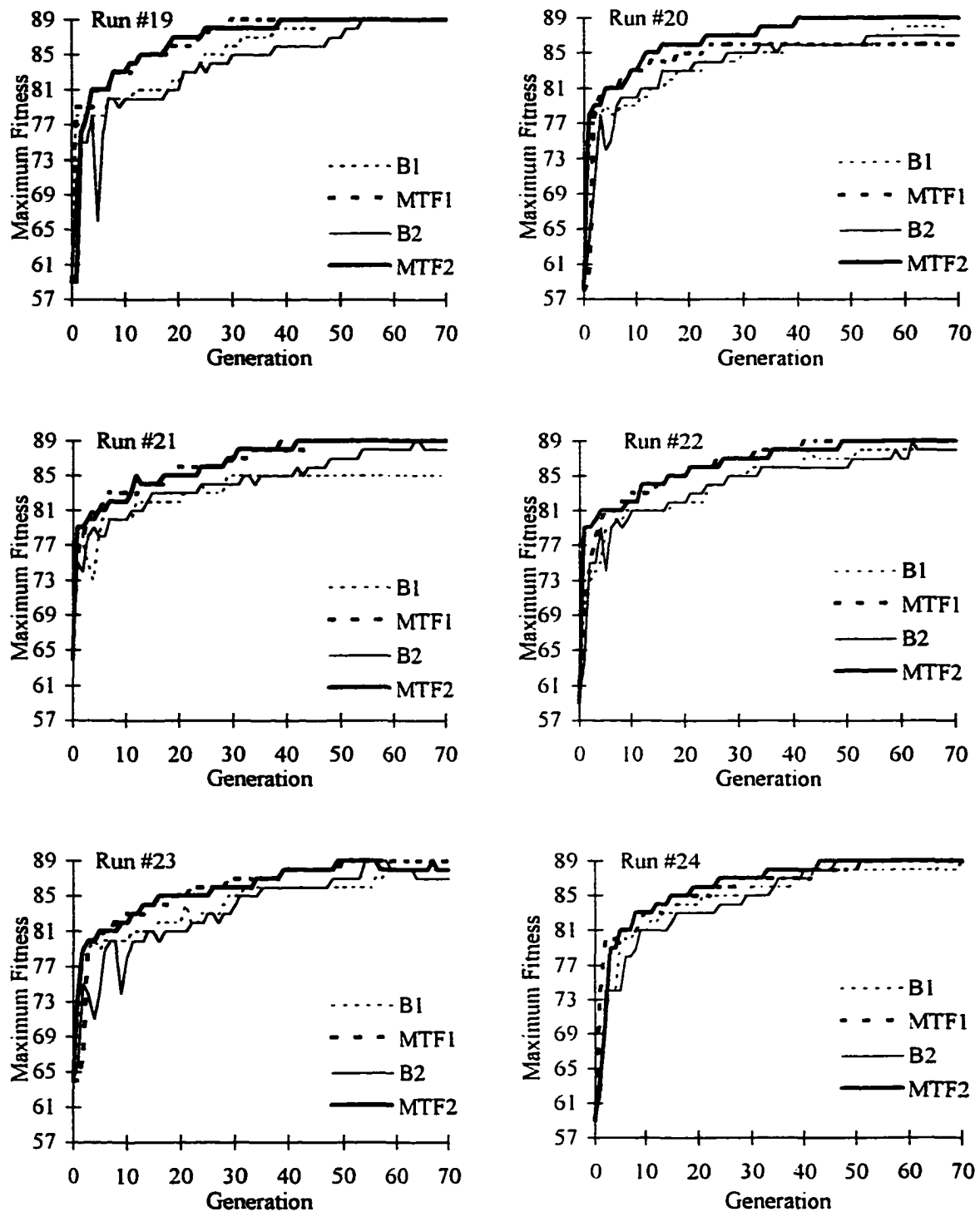


Figure C.5a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

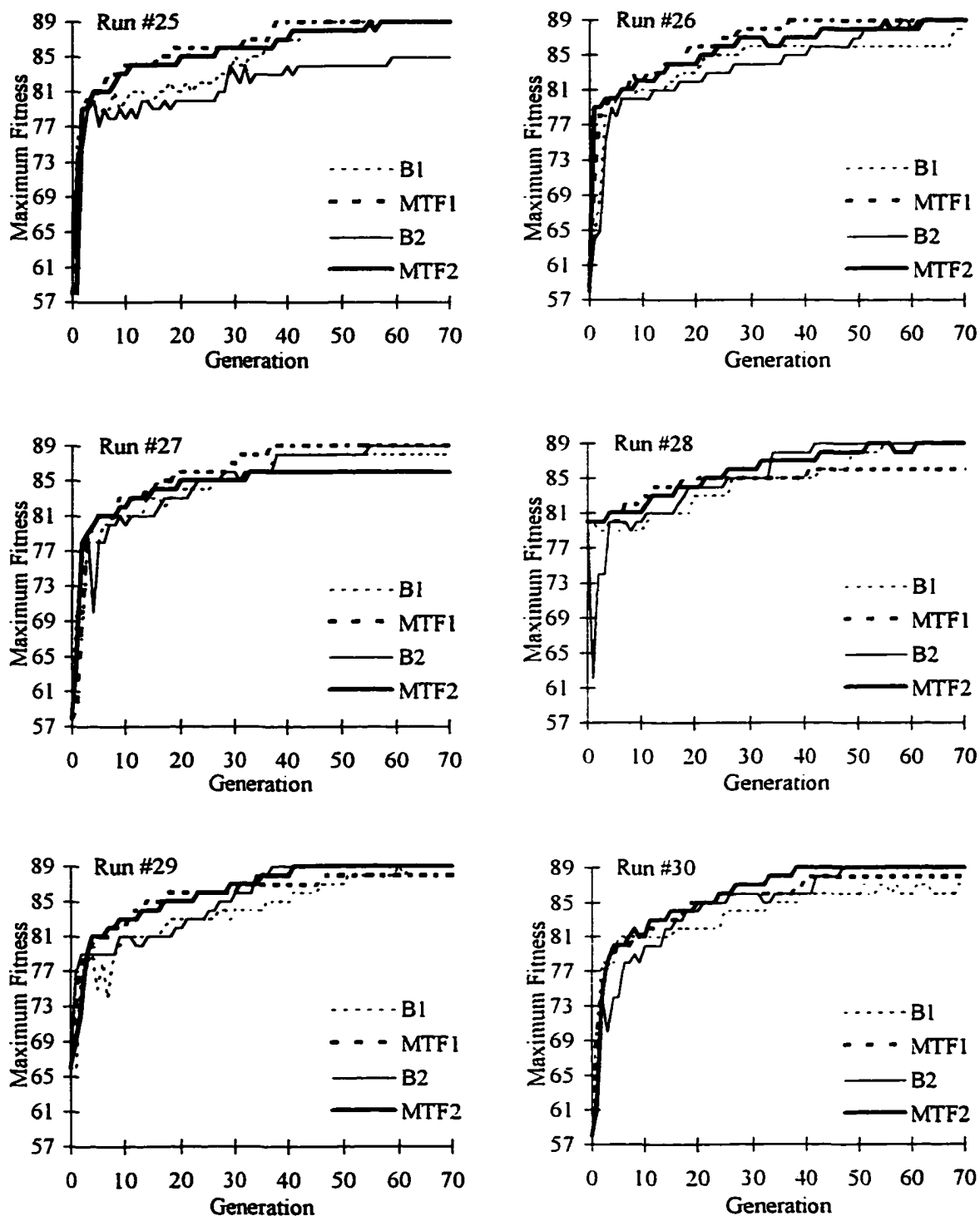


Figure C.5a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 64k. Identical initial population is basis for comparison.

Appendix C continued

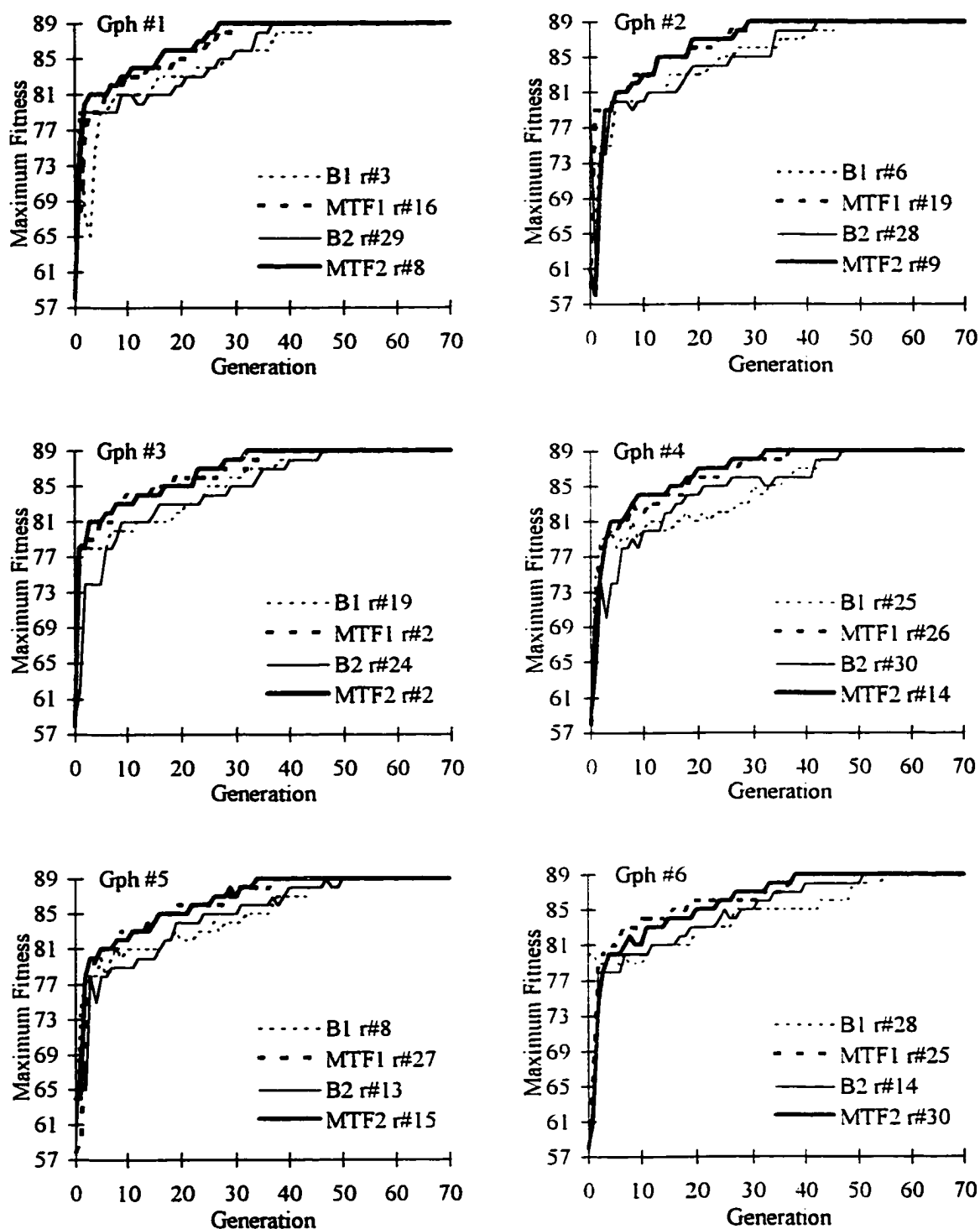


Figure C.5b: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

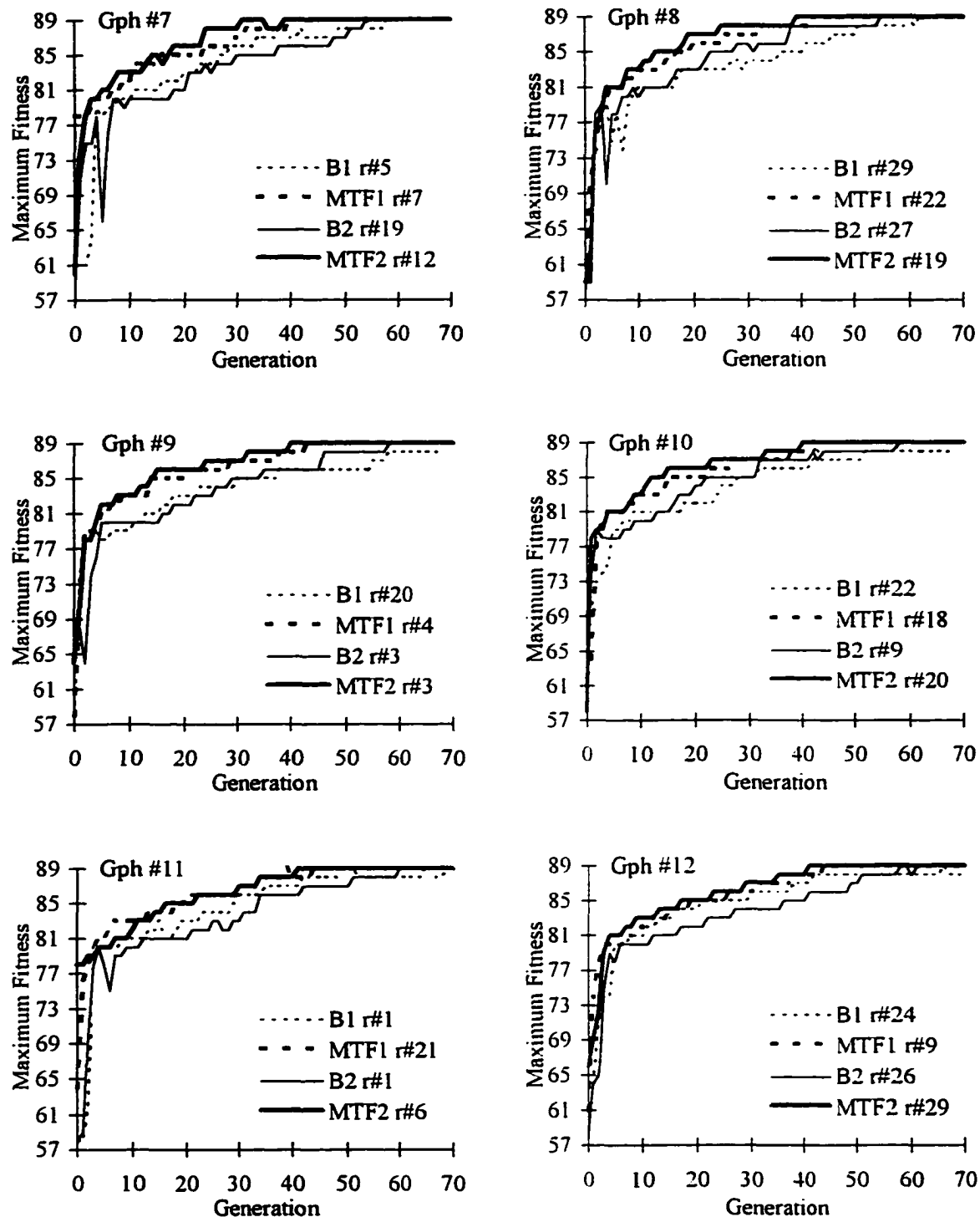


Figure C.5b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

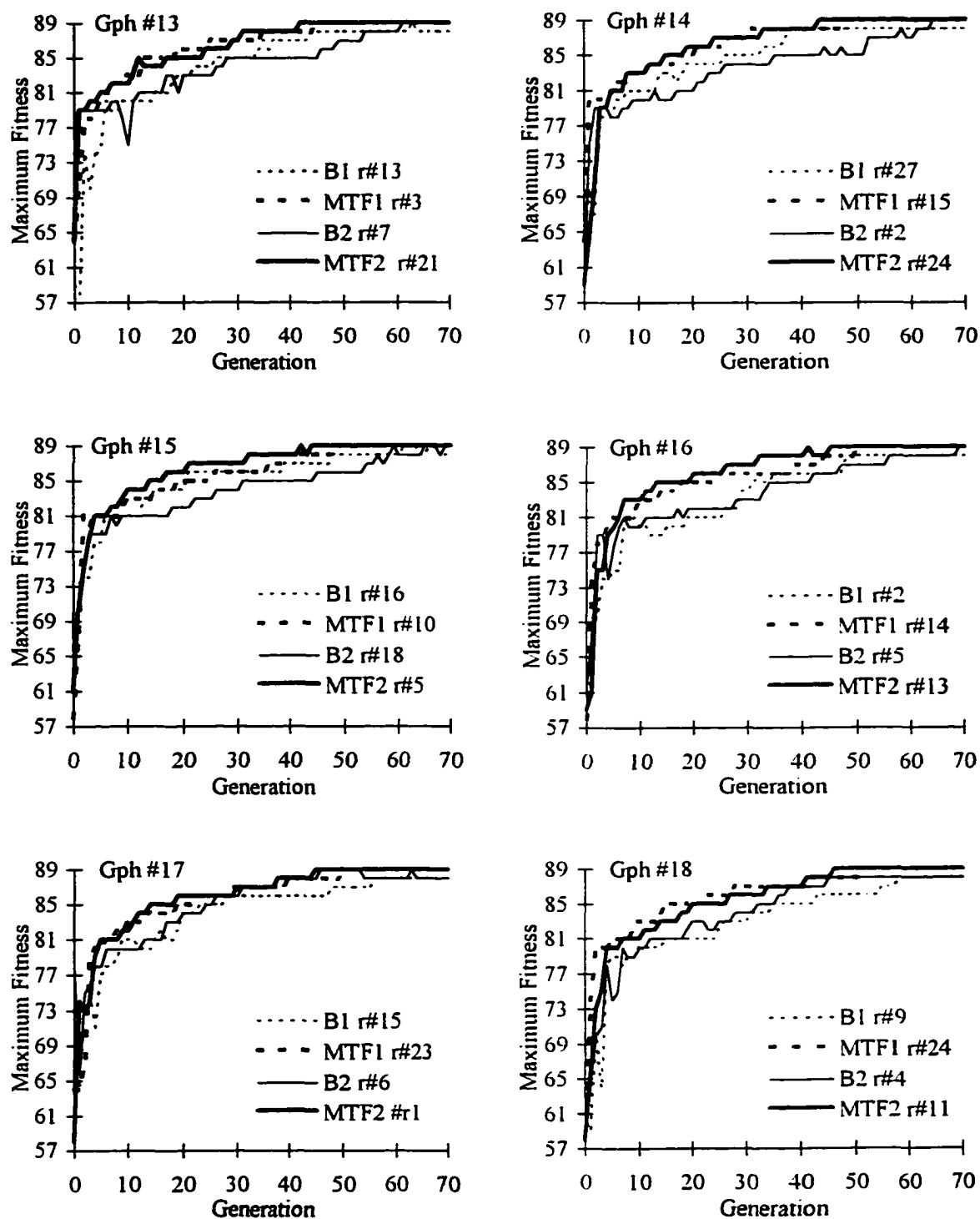


Figure C.5b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

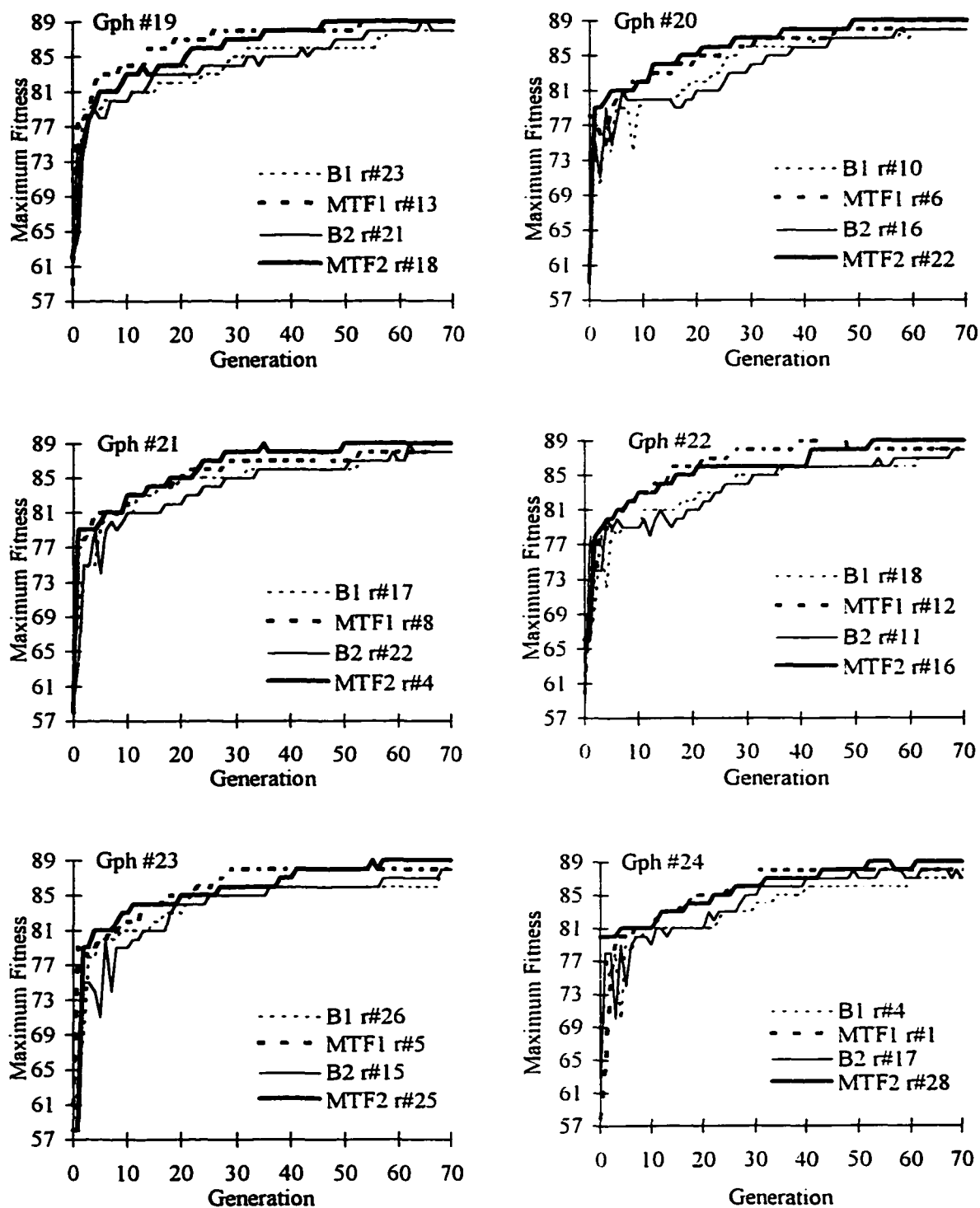


Figure C.5b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

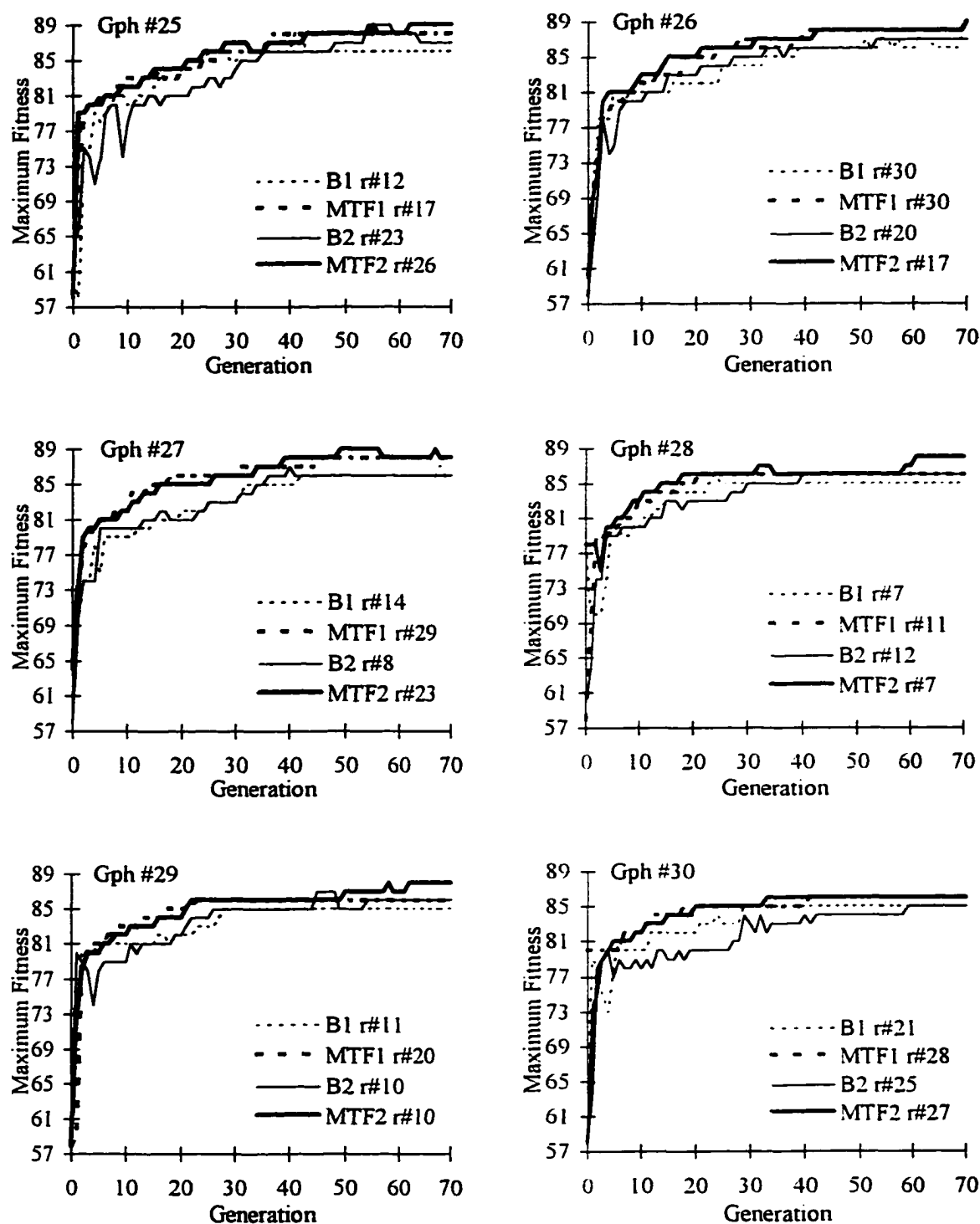


Figure C.5b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 64k. Evolutionary progress is basis for comparison.

Appendix C continued

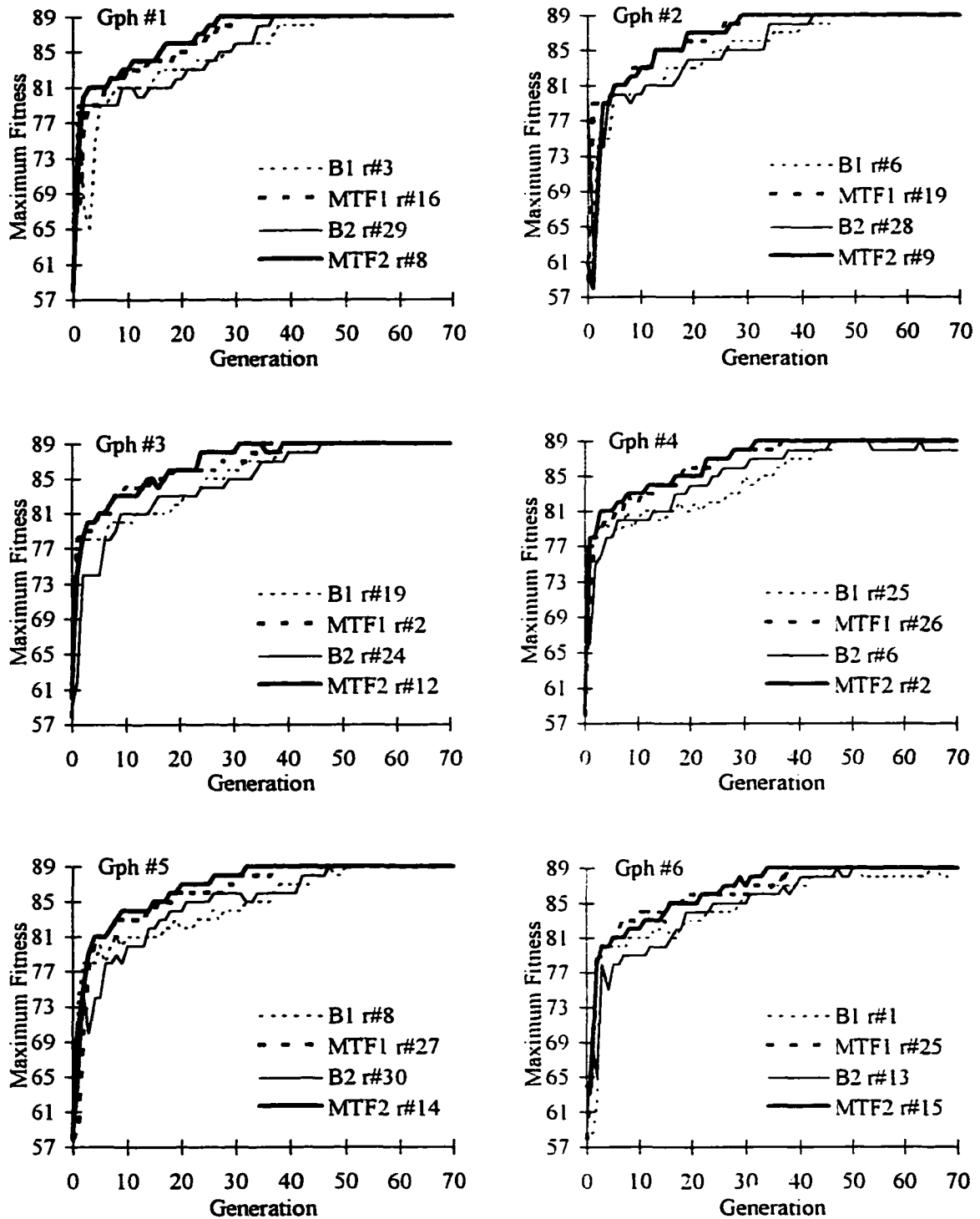


Figure C.5c: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B2, MTF2) and without competition (B1, MTF1) using a population of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

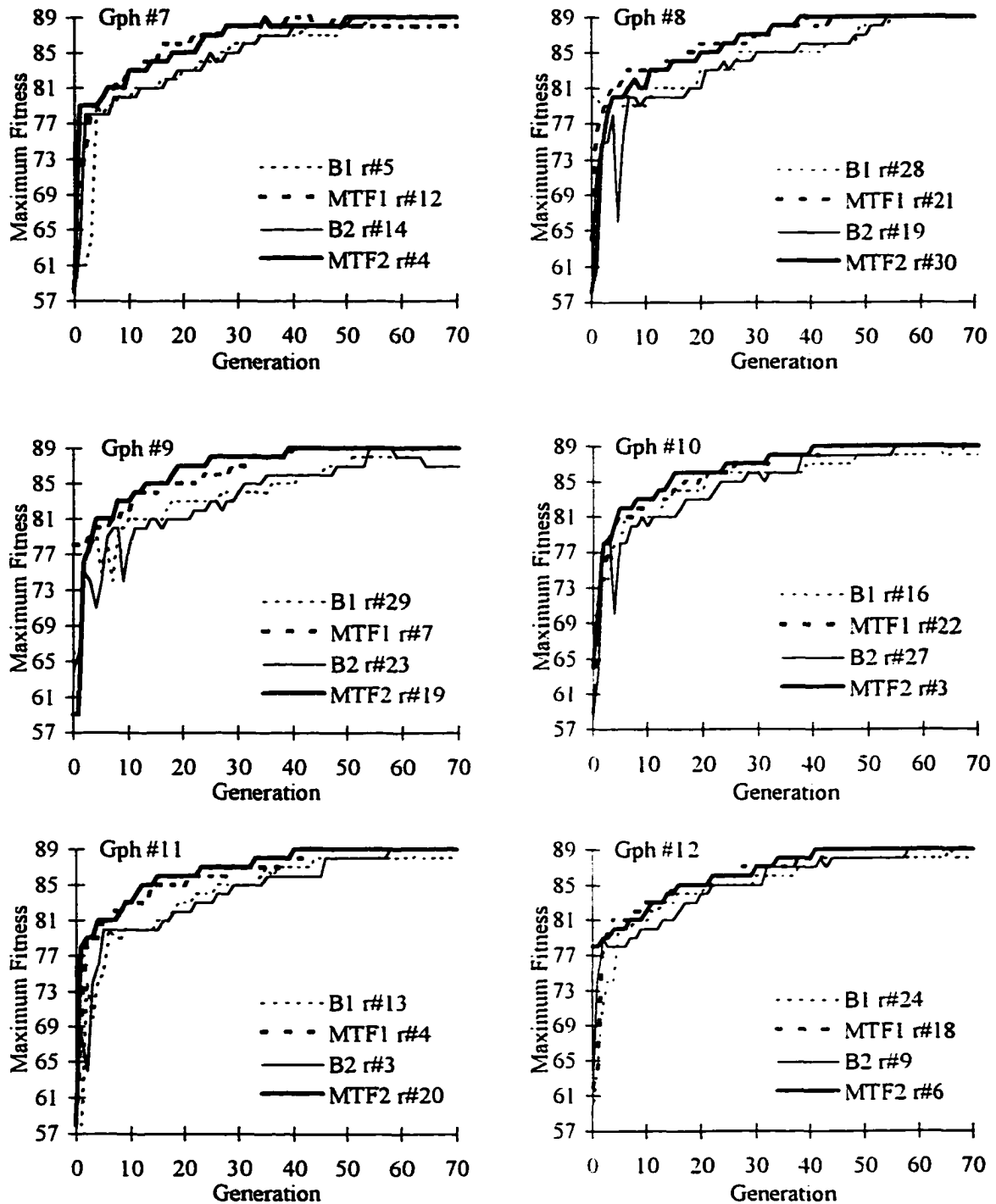


Figure C.5c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition ($B2, MTF2$) and without competition ($B1, MTF1$) using a population of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

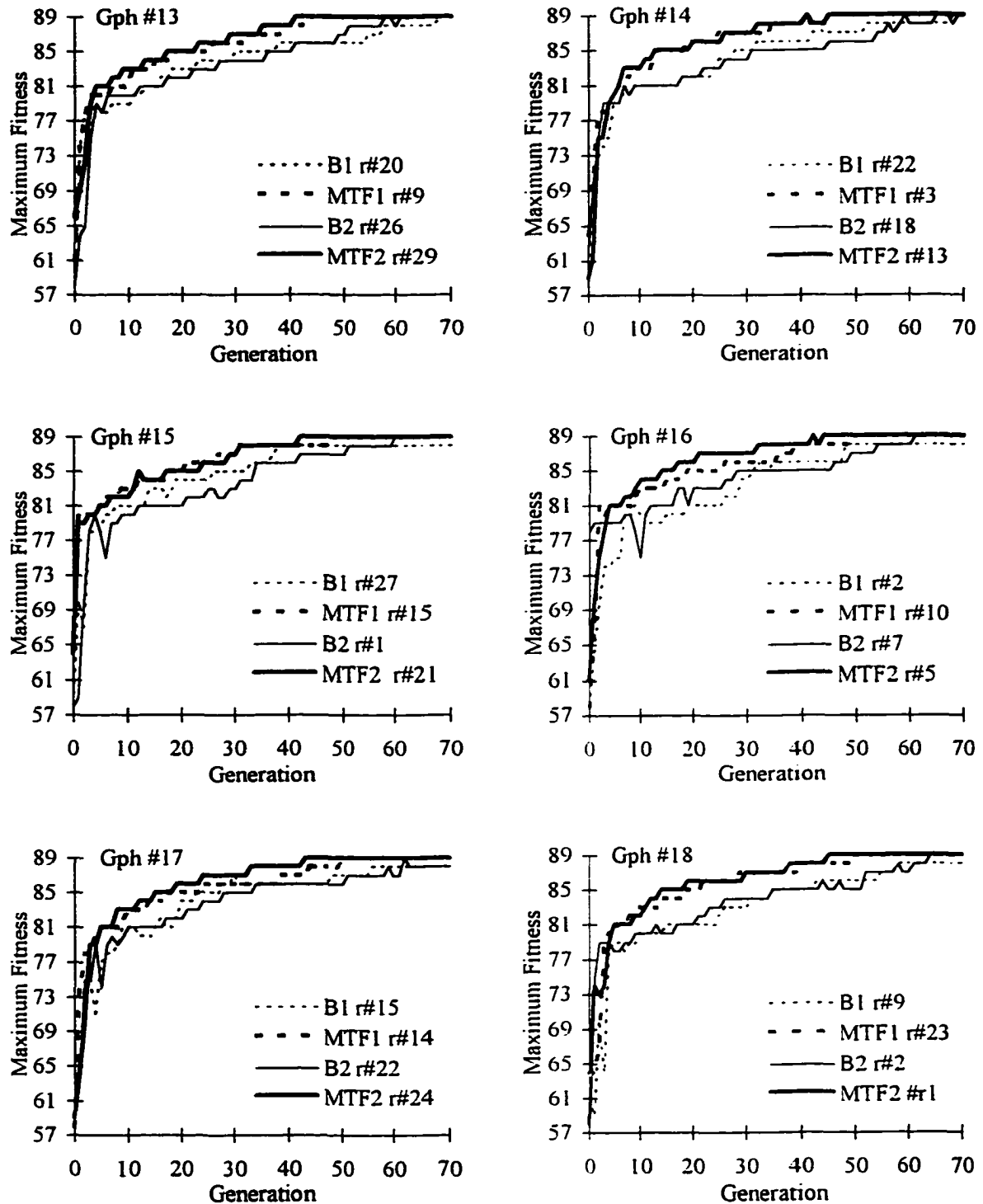


Figure C.5c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition ($B2, MTF2$) and without competition ($B1, MTF1$) using a population of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

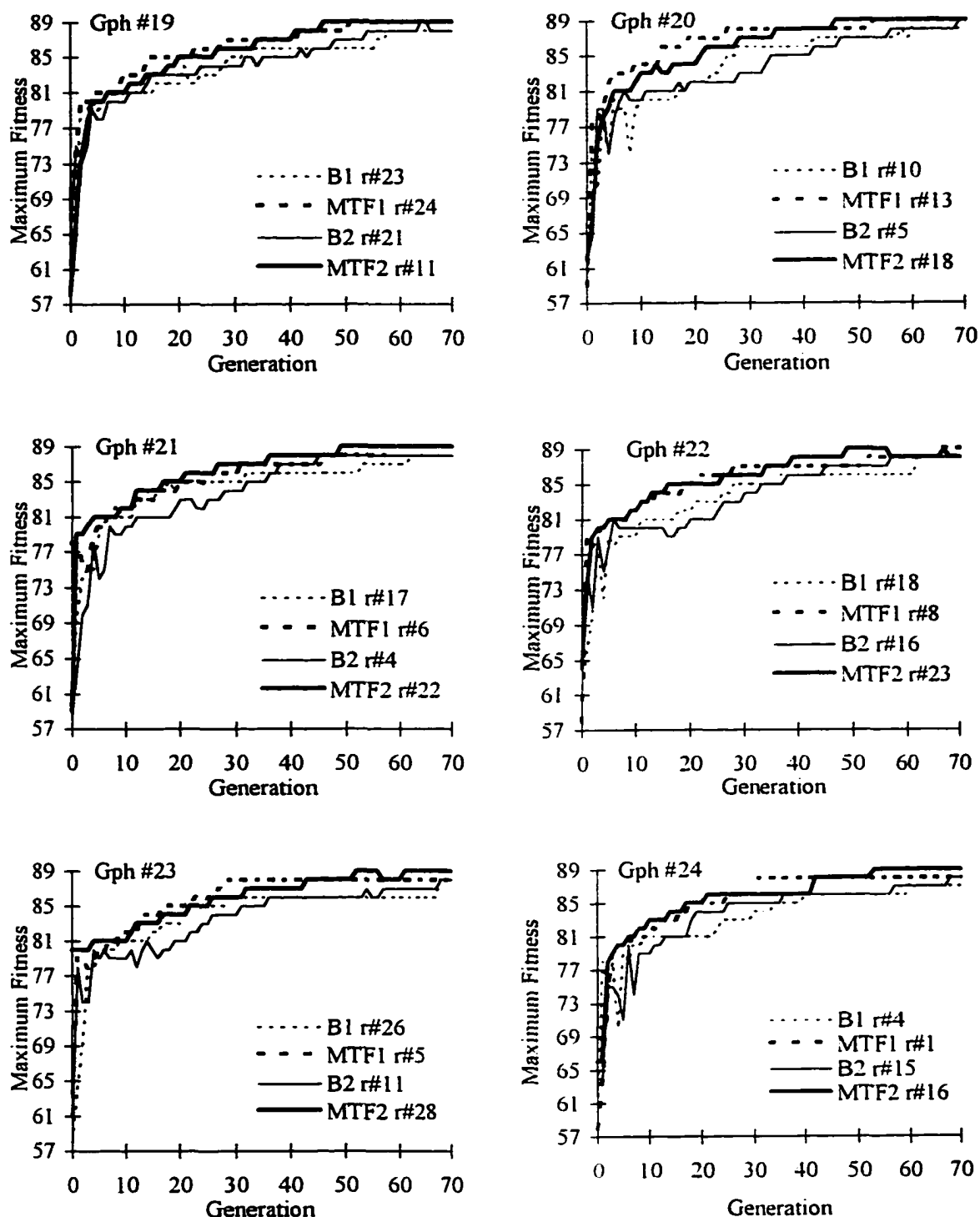


Figure C.5c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

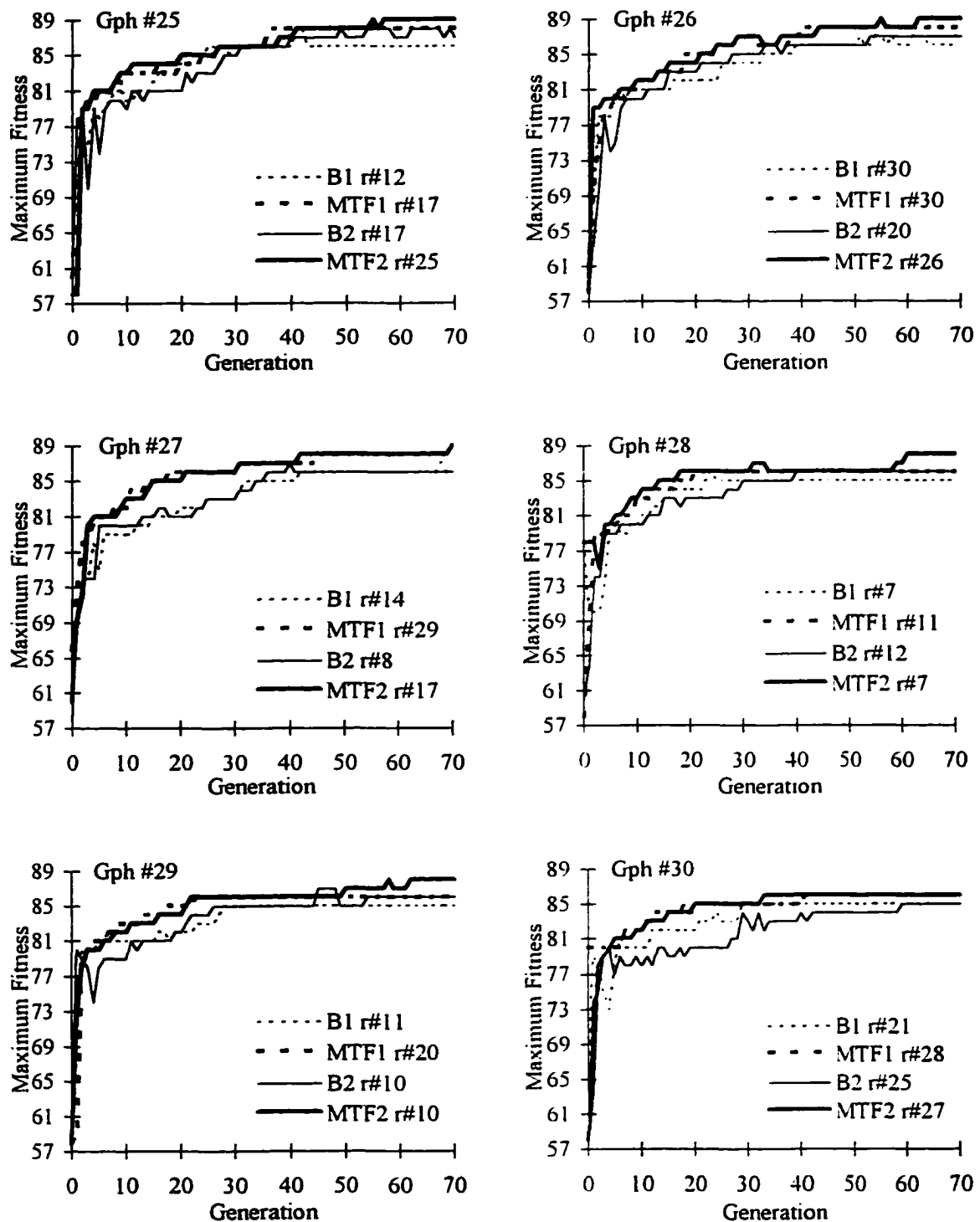


Figure C.5c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 64k. Location of optimal individual is basis for comparison.

Appendix C continued

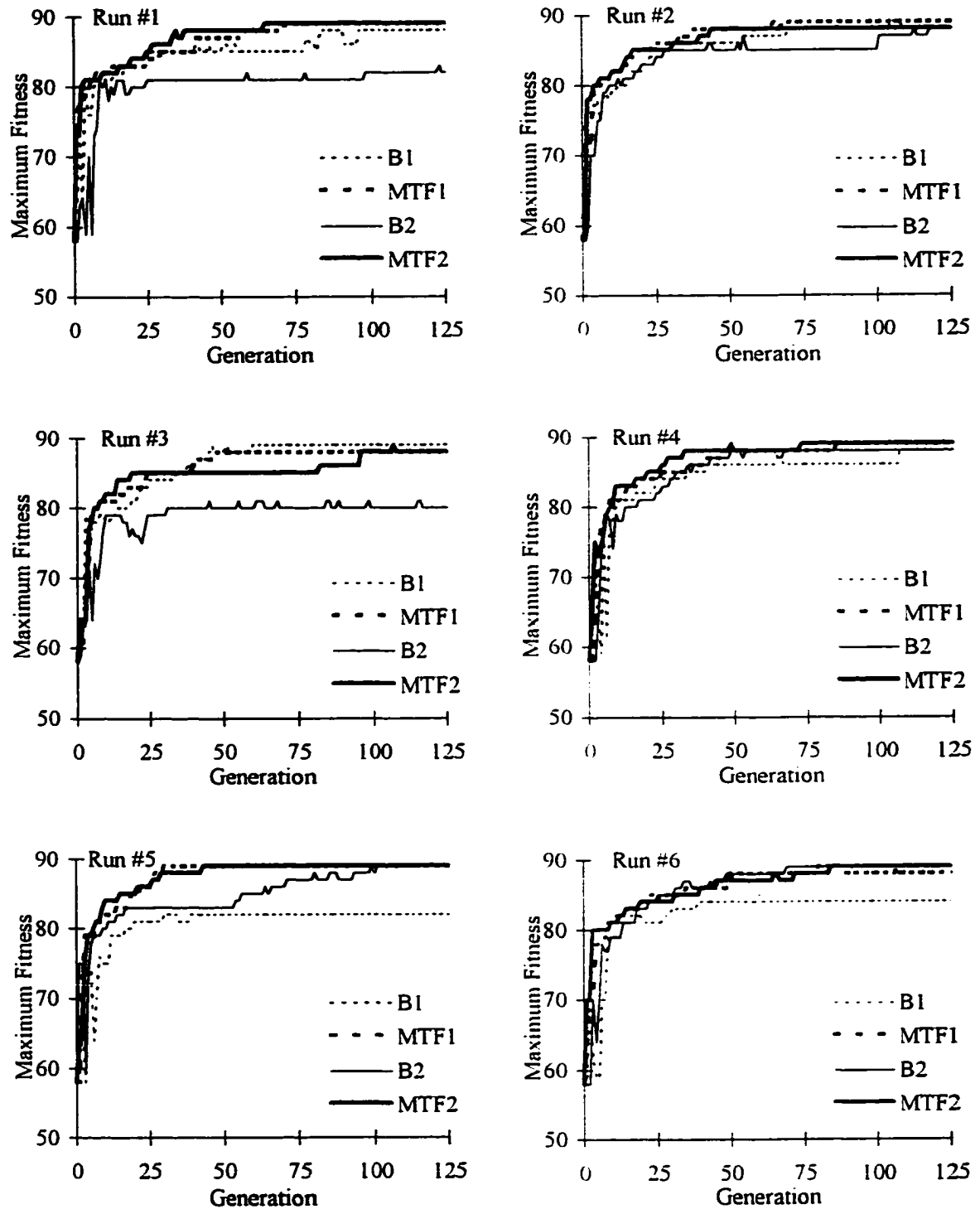


Figure C.6a: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

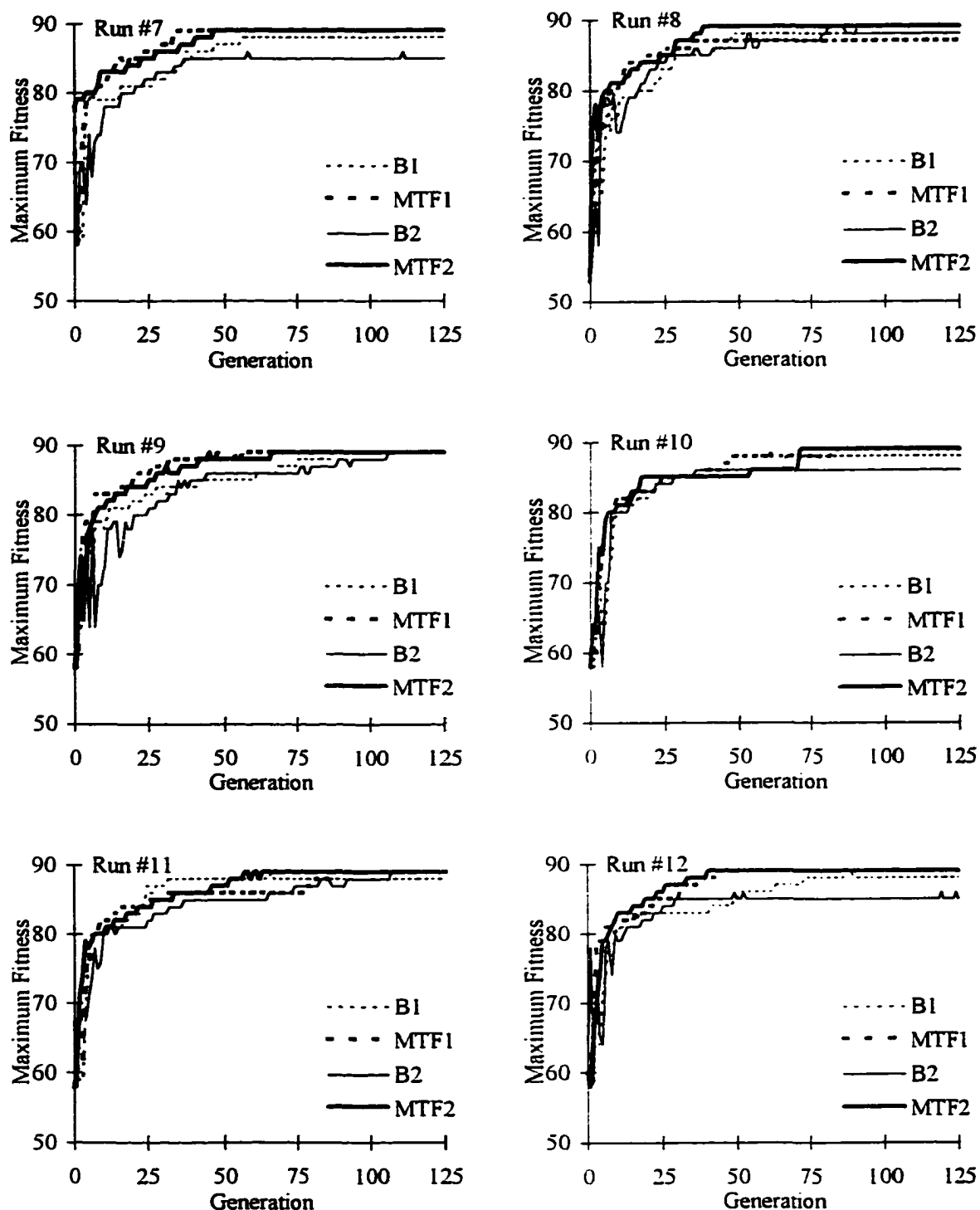


Figure C.6a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

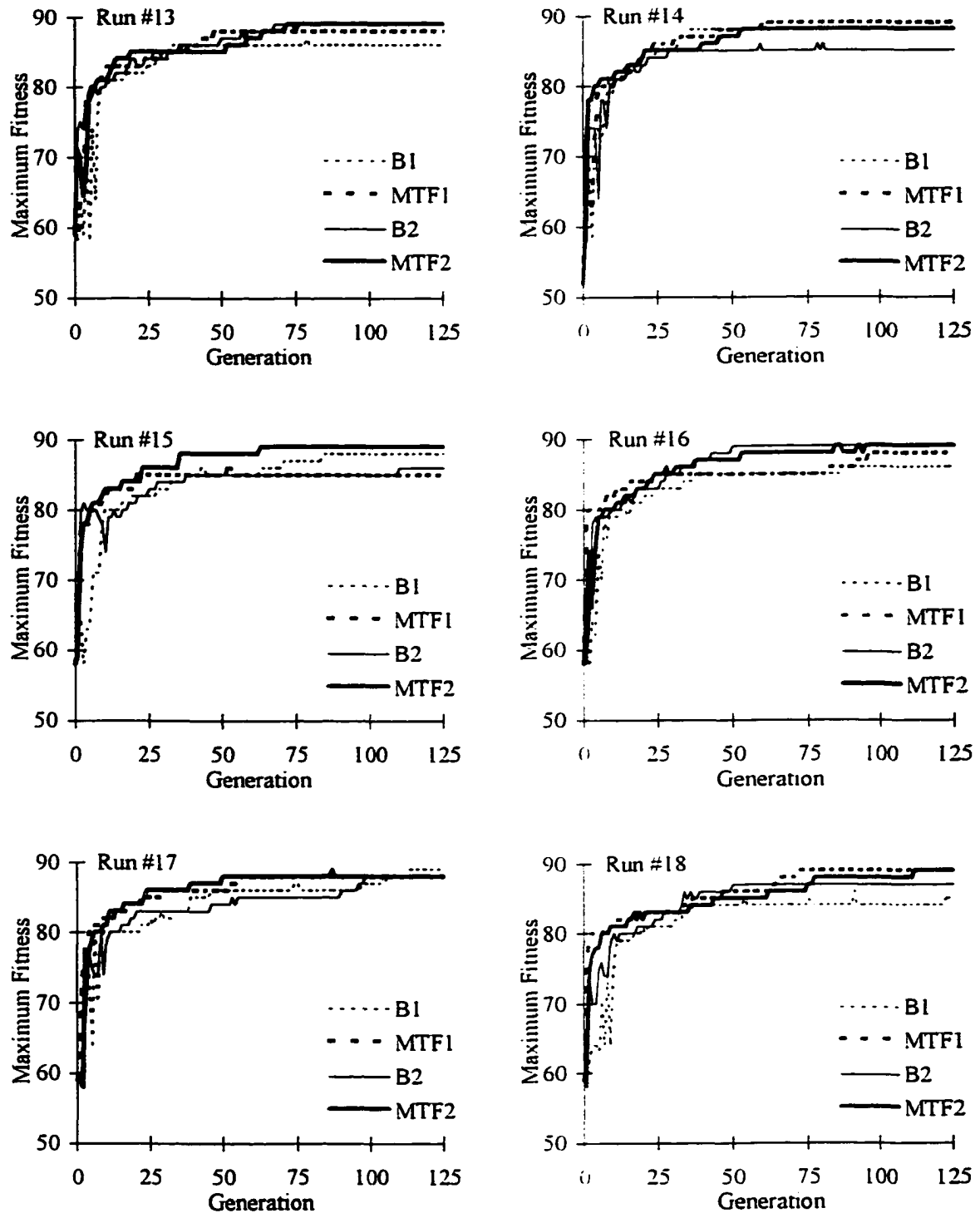


Figure C.6a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

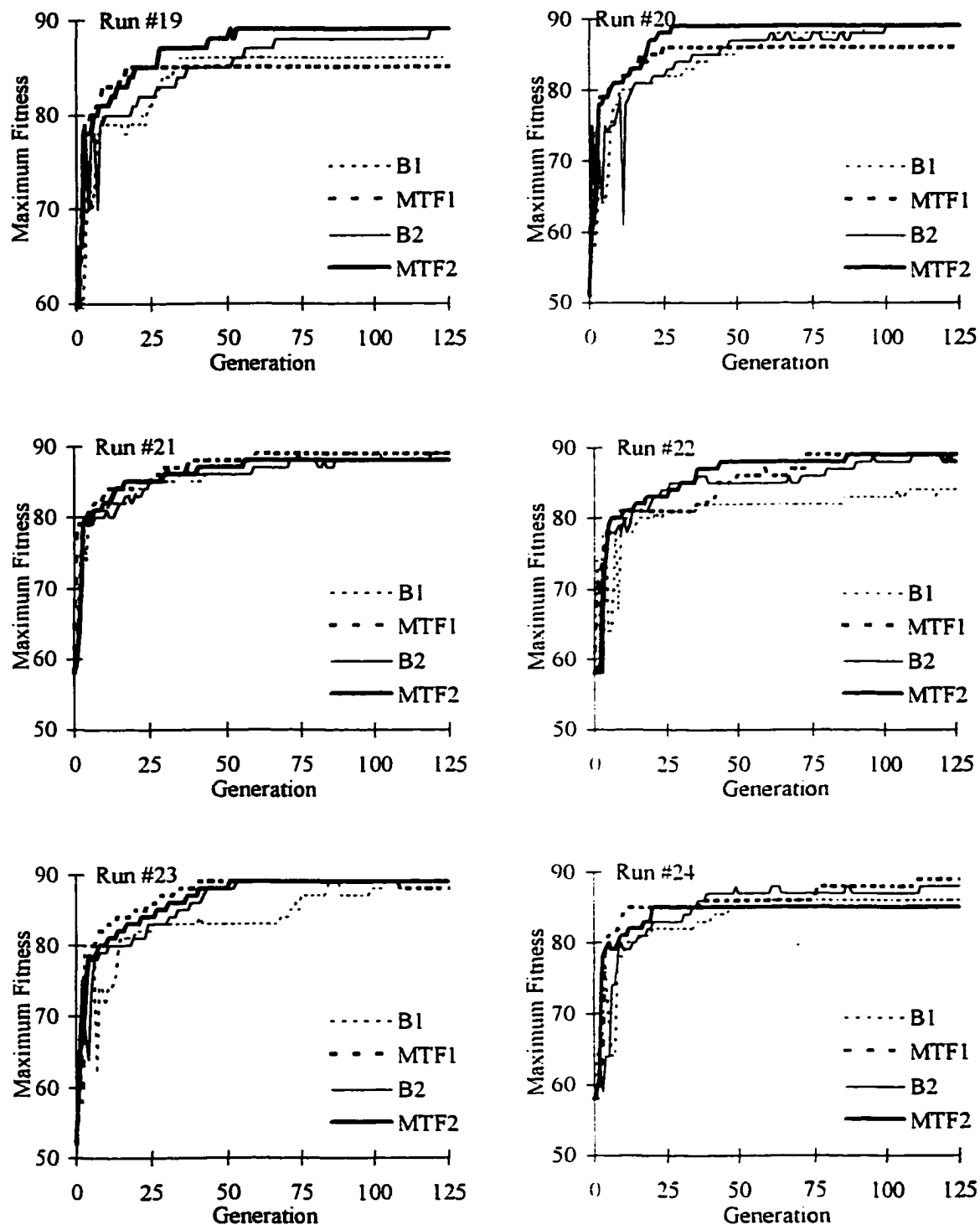


Figure C.6a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

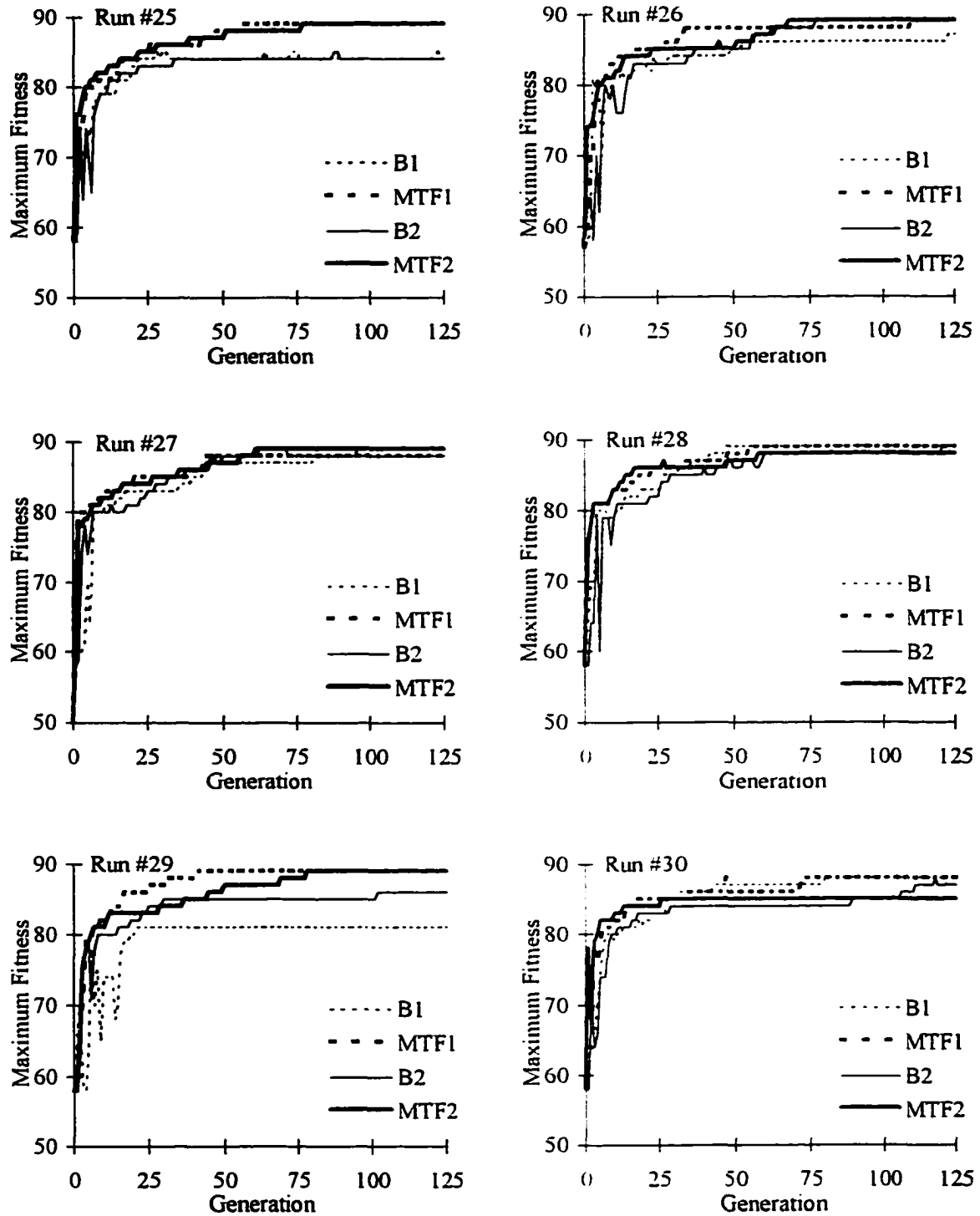


Figure C.6a continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B_2 and MTF_2) and without competition (B_1 and MTF_1) using a population size of 16k. Identical initial population is basis for comparison.

Appendix C continued

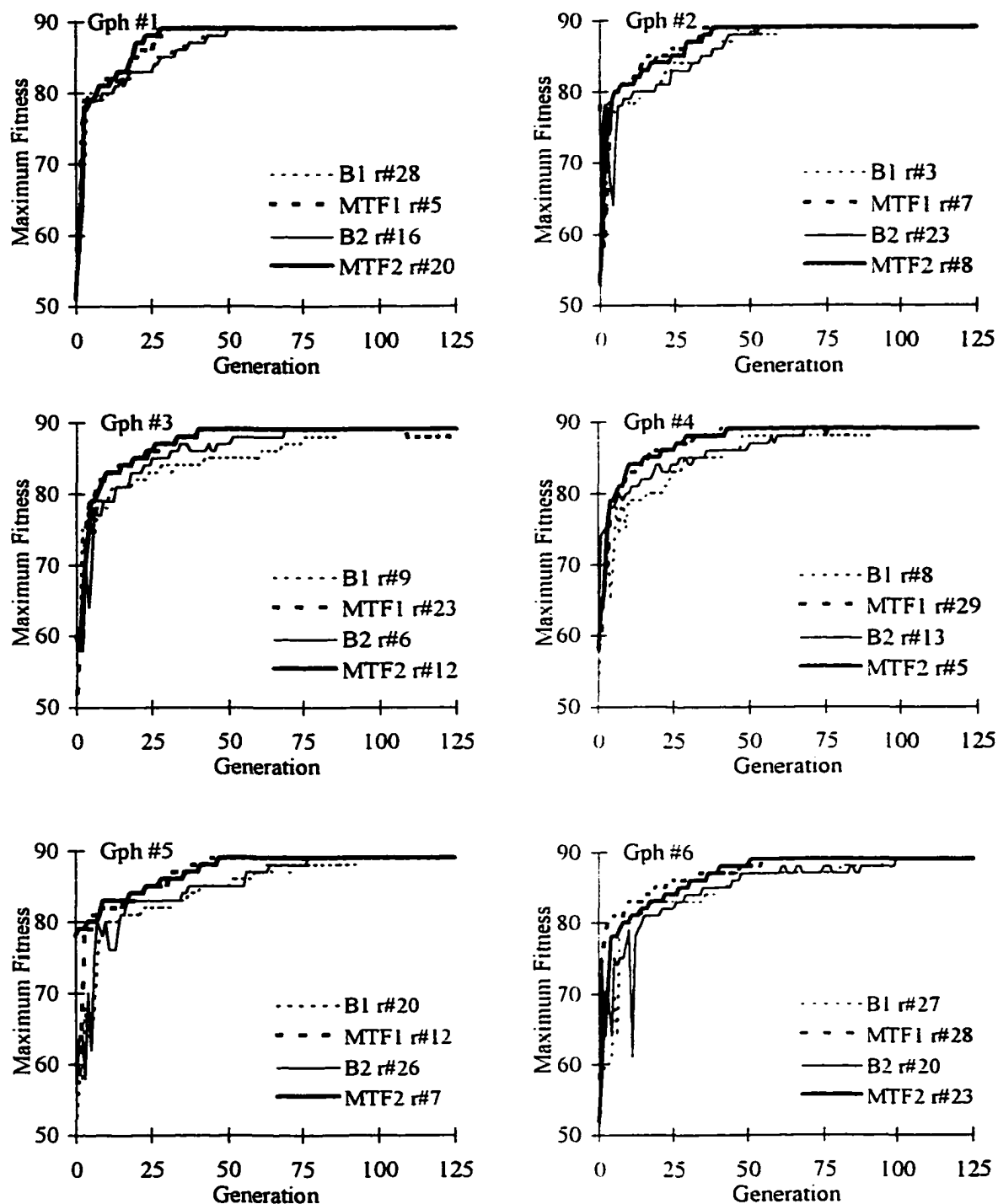


Figure C.6b: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

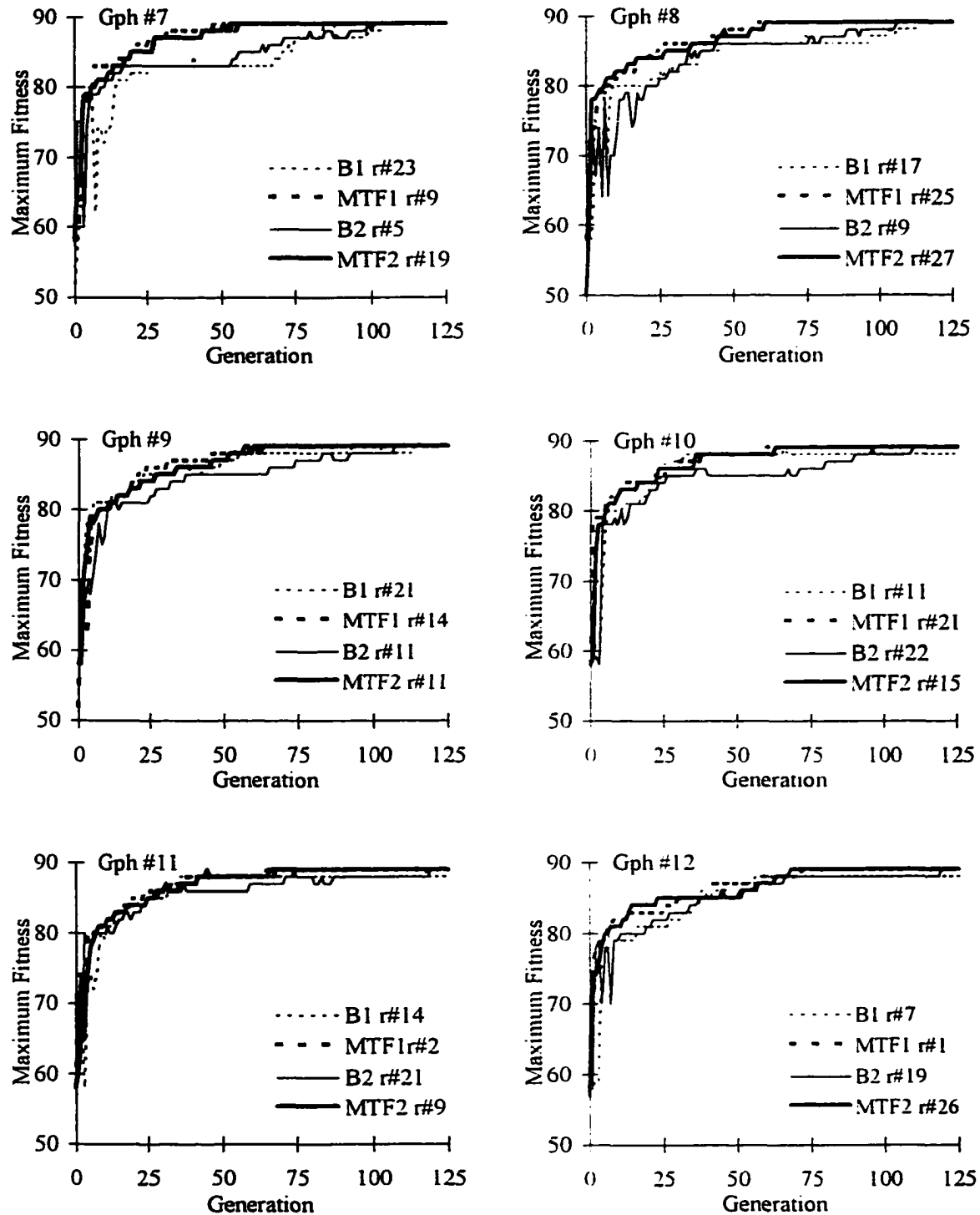


Figure C.6b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

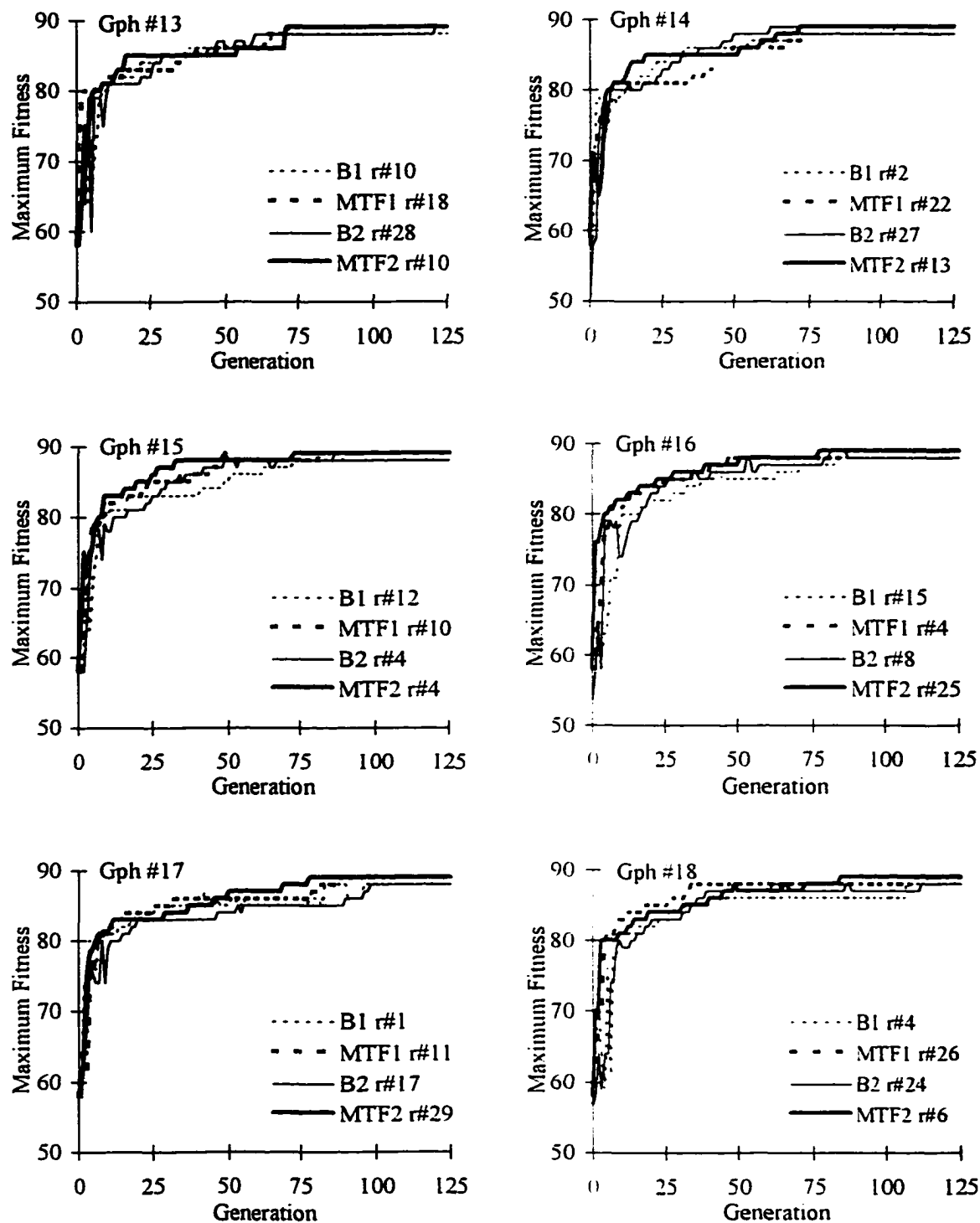


Figure C.6b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition (B2 and MTF2) and without competition (B1 and MTF1) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

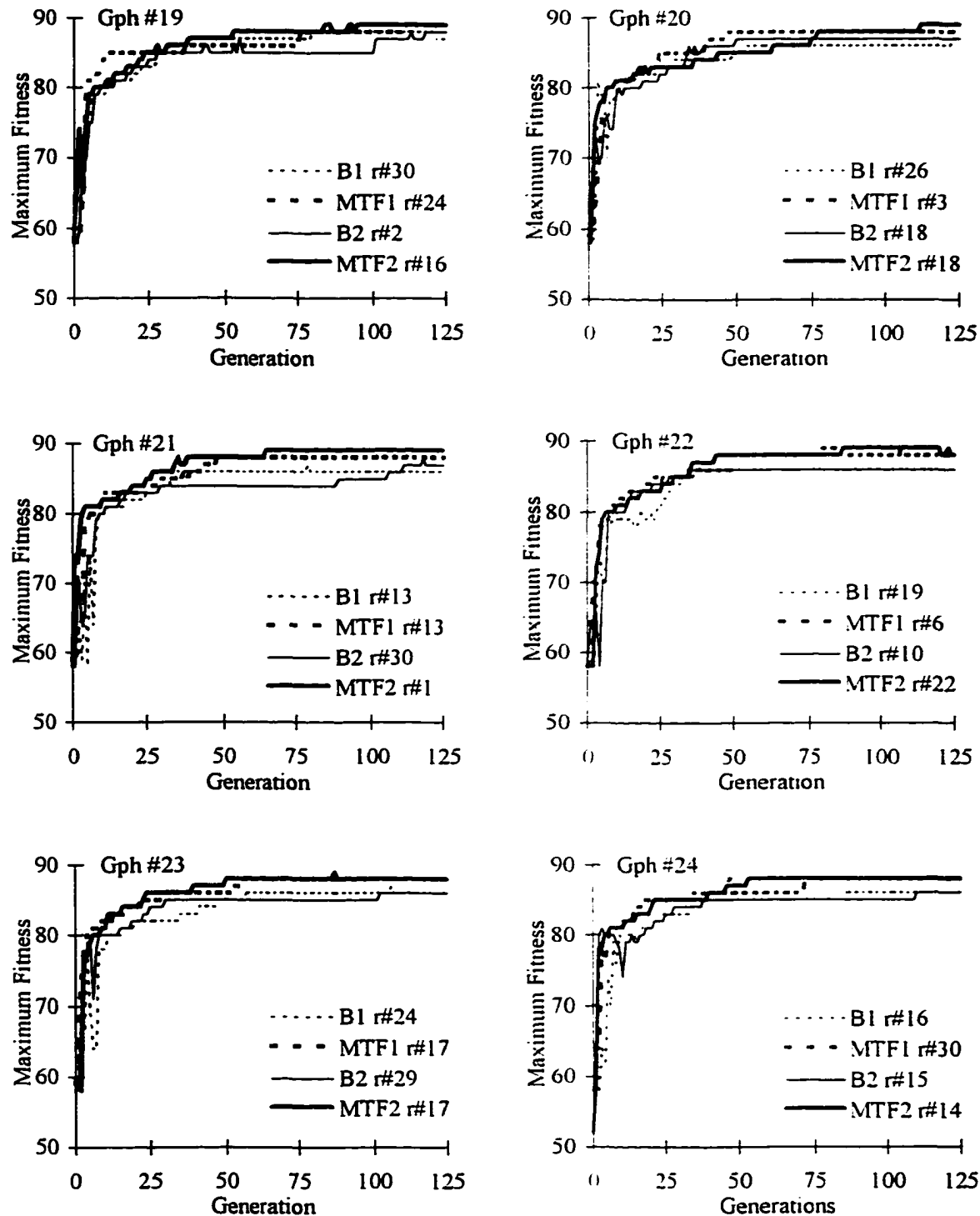


Figure C.6b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

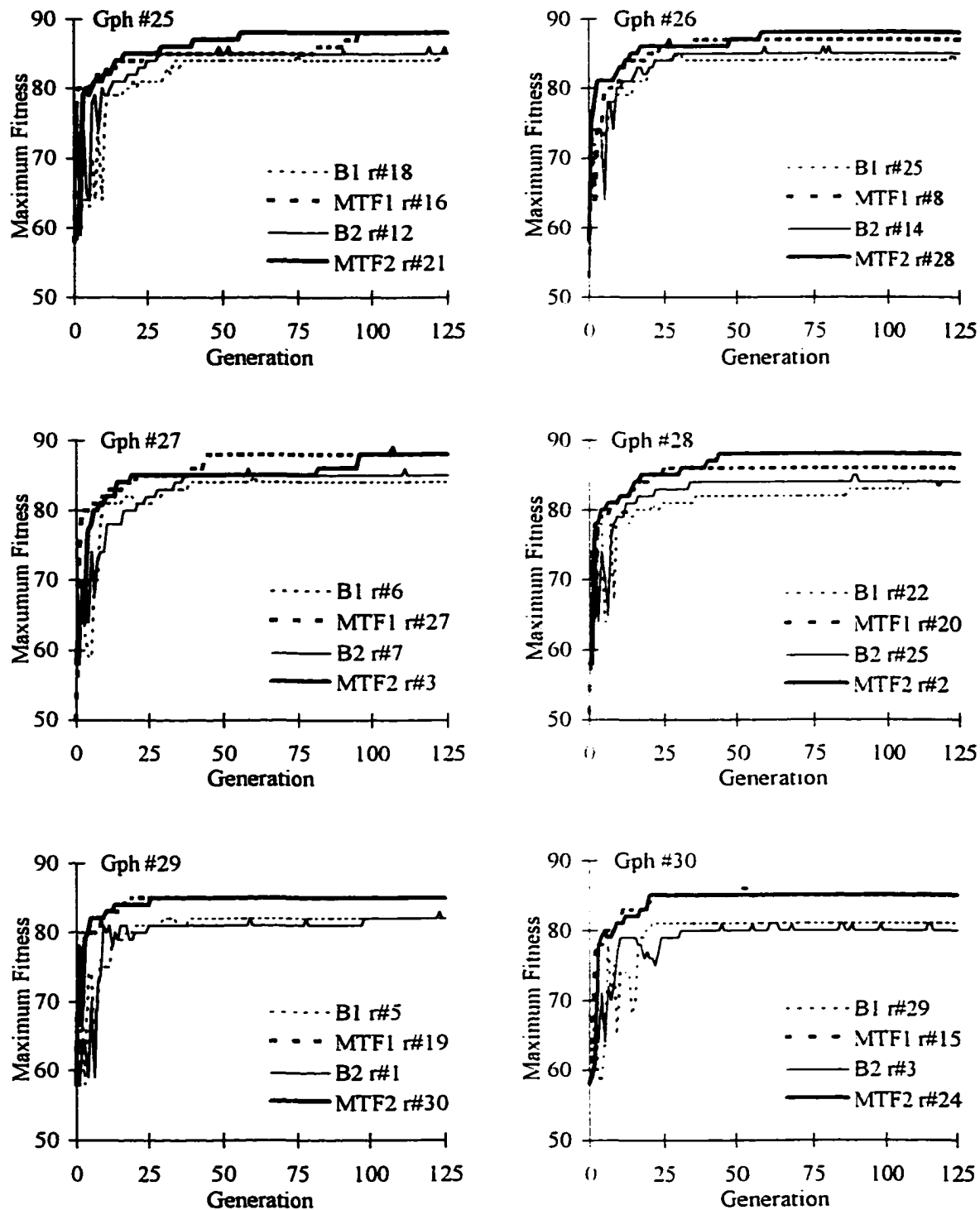


Figure C.6b continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), both with competition ($B2$ and $MTF2$) and without competition ($B1$ and $MTF1$) using a population size of 16k. Evolutionary progress is basis for comparison.

Appendix C continued

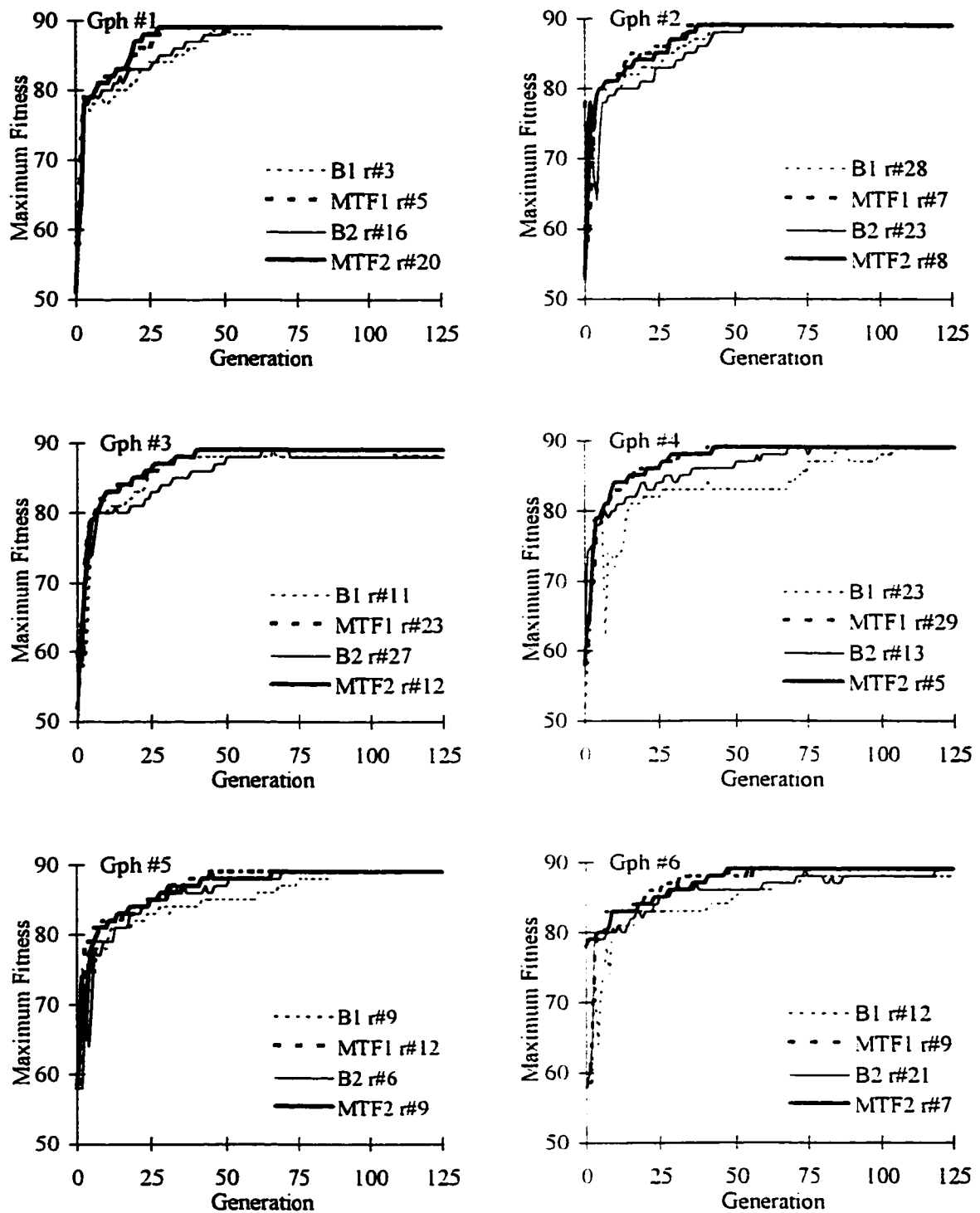


Figure C.6c: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B2, MTF2) and without competition (B1, MTF1) using a population of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

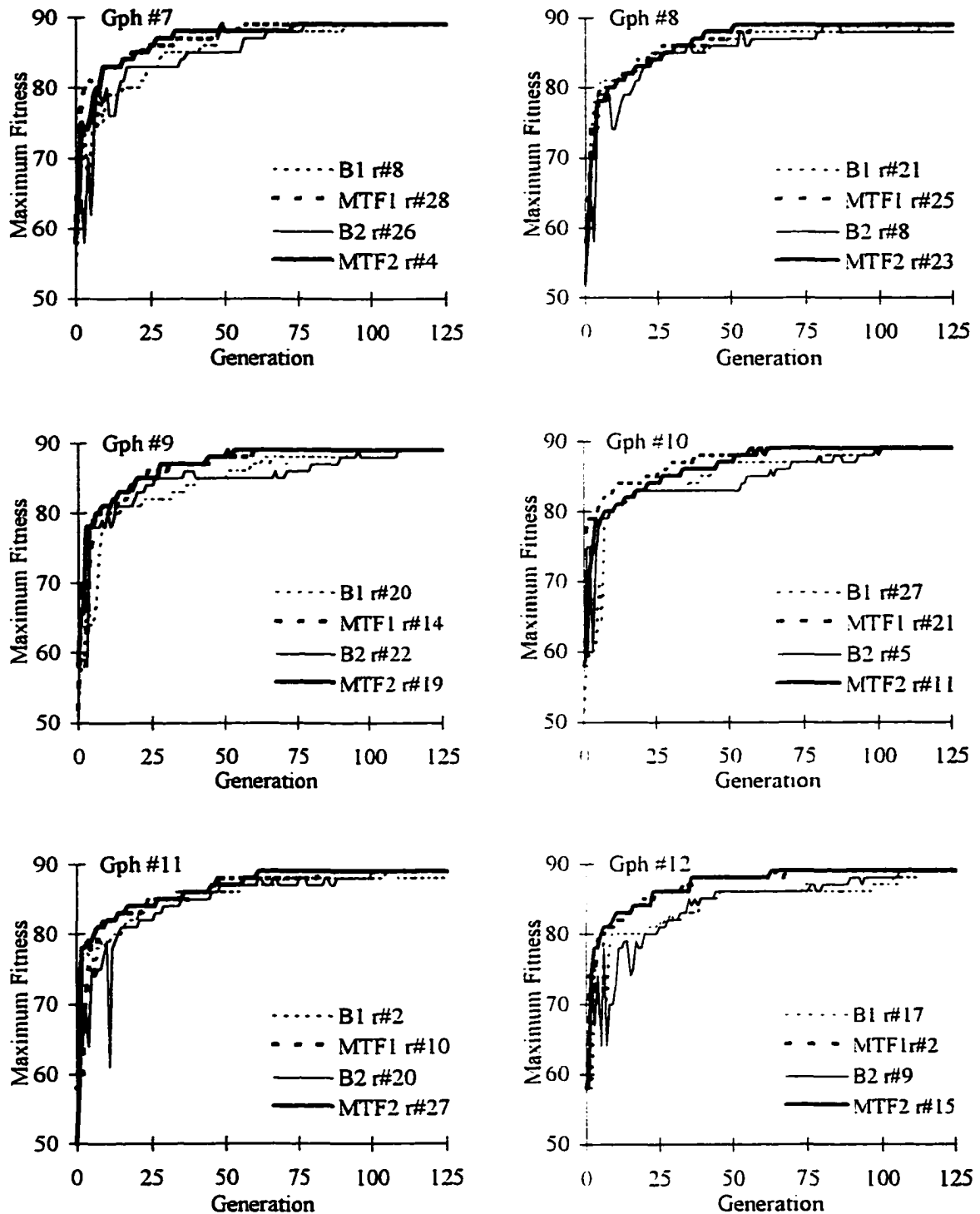


Figure C.6c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B2, MTF2) and without competition (B1, MTF1) using a population of 16k. Location of optimal individual is basis for comparison.

Appendix C continued

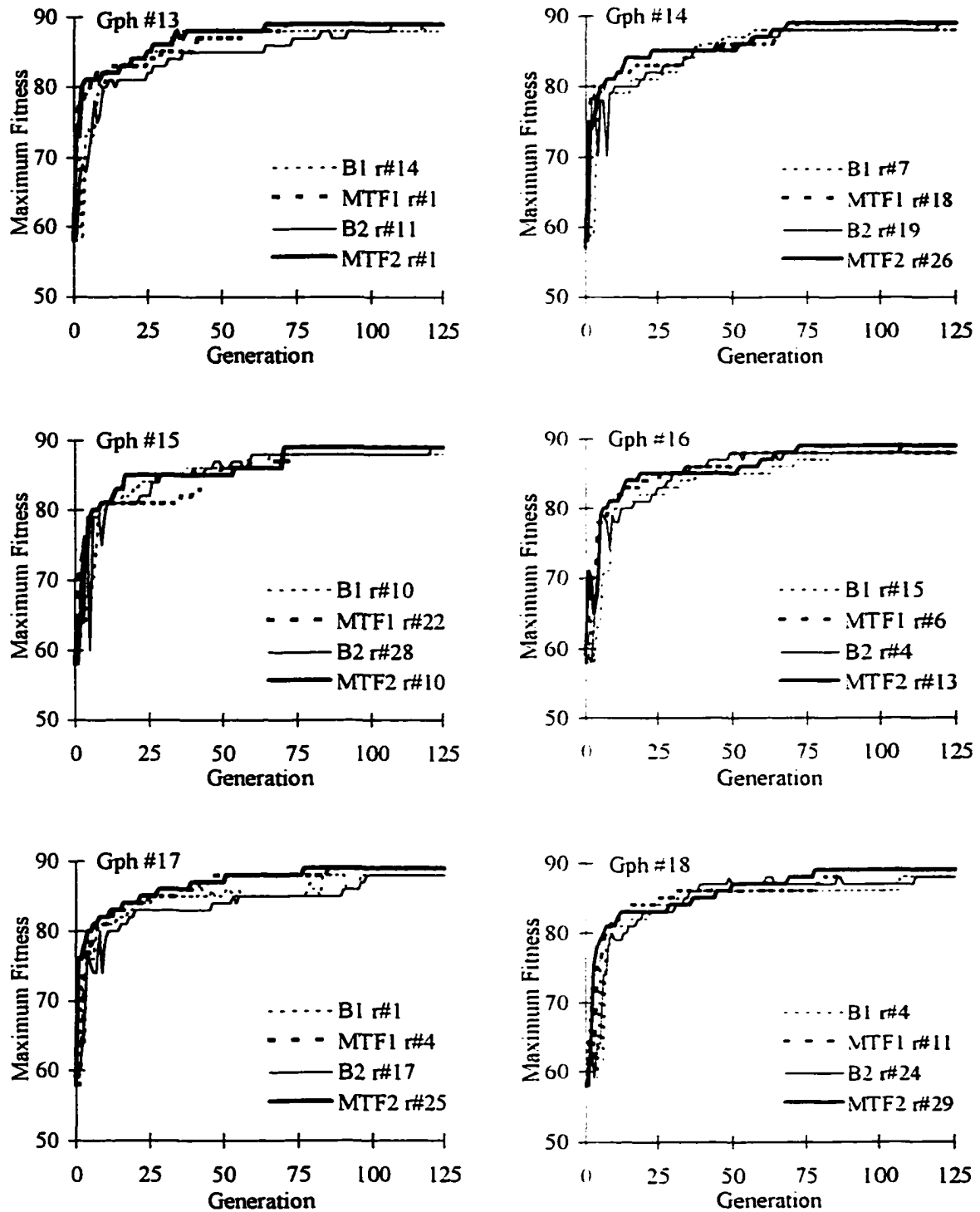


Figure C.6c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 16k. Location of optimal individual is basis for comparison

Appendix C continued

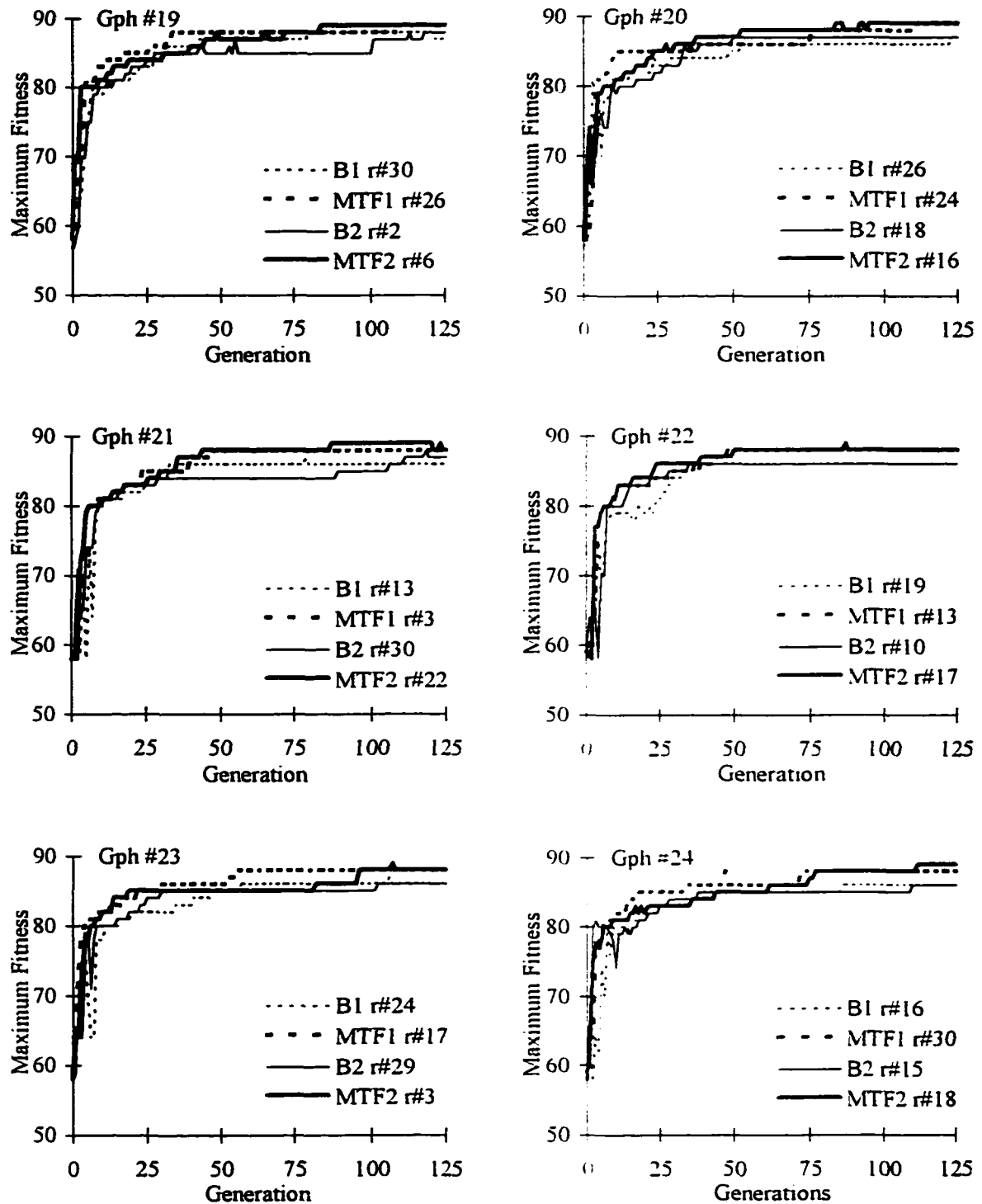


Figure C.6c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B2,MTF2) and without competition (B1,MTF1) using a population of 16k. Location of optimal individual is basis for comparison

Appendix C continued

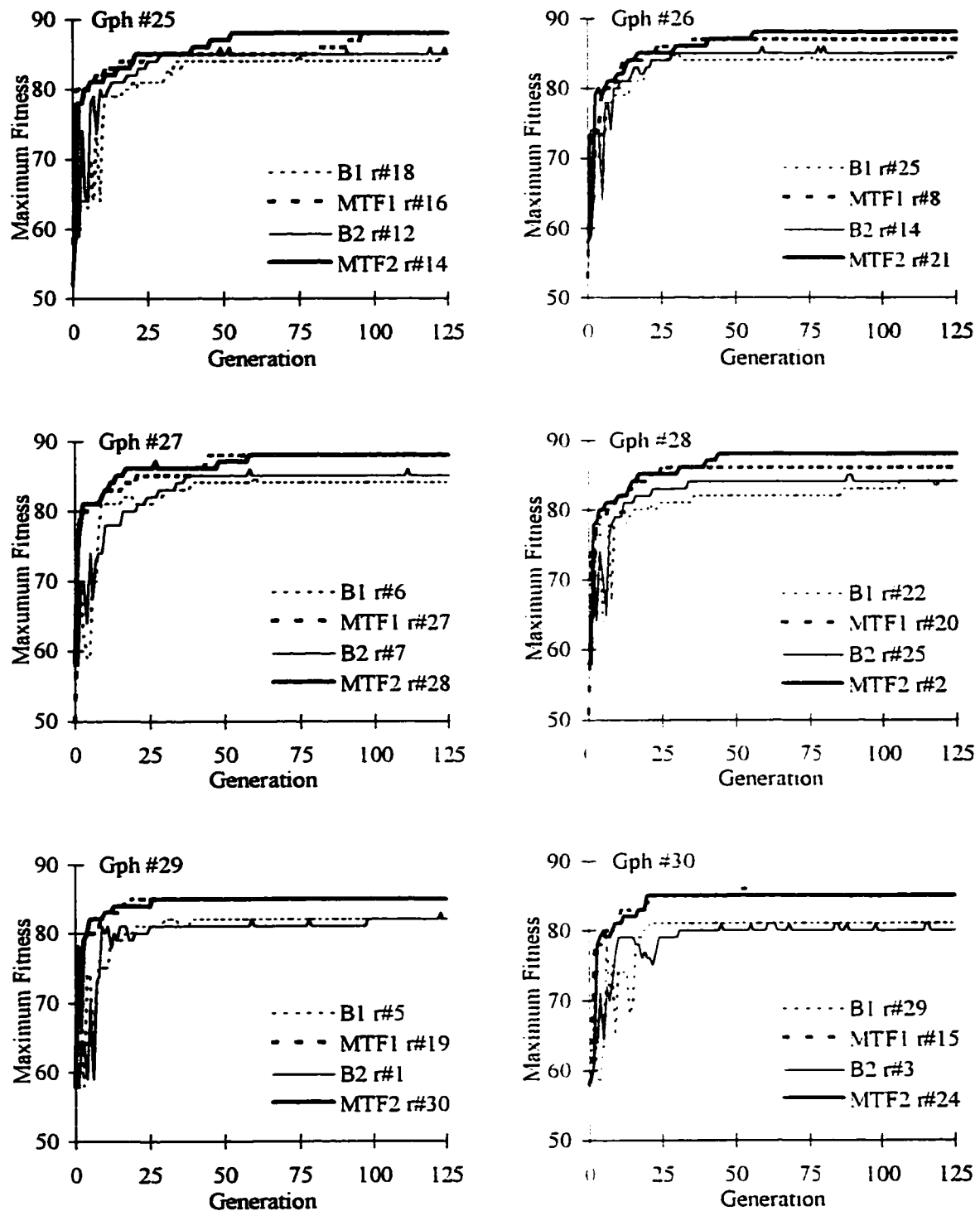


Figure C.6c continued: Comparison of benchmark (B_i) and benchmark with the MTF algorithm (MTF_i), with competition (B_2, MTF_2) and without competition (B_1, MTF_1) using a population of 16k. Location of optimal individual is basis for comparison.

Appendix D: Worksheets for language recognition problem

L2 worksheet											
recommendations				M1, M2, A1							
average machine size											
ranking				M2, A2, A1, M1, S1							
range				5.83 - 6.65							
F=	0 - 17	lo		Gav=	6.23 - 16.6	lo					
	1 - 31	hi			8.43 - 33.24	hi					

Appendix D continued

L2		continued										
Gav-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	1.6----->			1.4	1.5	1.7	1.4	1.1----->			~
*	M2	1.8----->			1.6----->		1.9	1.4	1.1	1.2	1.3	~
*	A1	1.6----->			1.5	1.4----->		1.1	S	1.2----->		~
*	A2	1.2----->		1.6	1.4----->			1.2	1.1	1.2	1.1	
	S1					1.1	1.4	1.2	~	1.1----->		
?s	S2	1.1----->			~	~	~				1.1----->	
*	B2	~	~	~	1.3----->			1.2	1.1----->		1.2	
Gmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	1.6----->							1.5	1.4	1.3	1.2
*	M2	1.1	1.2	1.3	1.5----->			1.6	1.5	1.4	1.3----->	
*	A1	1.6----->				1.5----->			1.4	1.3----->		
*	A2	1.2----->			1.3----->			1.4	1.3	1.2----->		
	S1						~	~	1.1	~	1.1----->	
s	S2	1.1----->				~	~	~		~	~	
s *	B2	~	~	~	~	1.1----->						1.2
probability of fewer generations-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	19----->									18	16
*	M2	18----->									17	18
*	A1	23----->								22----->		23
*	A2	8.3----->						8.4	8.2	7.7----->		
*	S1	9.1----->				9	9.5	9.8	9.3	9.5	12	
	S2	~	~	~	~	~	~	~				
	B2											1.9

Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		1.5----->			1.4----->		1.6	1.3	~	1.1	~	~
* M2		1.5----->			1.4----->		1.6	1.2	~	~	1.1	
* A1		1.5----->			1.4----->			~	~	1.1----->		~
s *	A2	~	~	1.4	1.2----->			~	~	1.1	~	
	S1					1.1	1.3	1.1	~	1.1----->		~
	S2									~	~	~
	B2					1.2----->		~	~	~	1.1	
Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		1.5----->			1.6	1.5----->				1.4	1.3	1.2
* M2		~	1.1	1.2	1.3----->		1.4----->			1.3	1.2----->	
* A1		1.5----->						1.4	1.3	1.2----->		
* A2		~	~	1.1	1.2----->			1.1----->				
	S1							~	~	~	~	1.1
	S2				~							
	B2					~	~	~	~	~	~	1.1
probability of less procesor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		13----->								14	13	12
* M2		2.5----->			2.4----->			2.5	2.4	2.1	~	~
* A1		18----->								16----->	17	
	A2											
* S1		4.3----->			4.2----->			4.5----->		3.9	3.6	4.9
	S2											
	B2											

Appendix D continued

L2c											
worksheet											
recommendations				M1, A1							
average machine size											
ranking		A1, M1, M2									
range		5.7 - 6.35									
F=	0 - 16	lo	Gav=		6.5 - 15.35		lo				
	1 - 27	hi			7.77 - 32.99		hi				

~: same as B1											
worse than B1											
*: recommended											
s: similar to B1											
?: not conclusive											

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
*	M1	~	~	~	~	~	~	1.2	1.1-----> 1.4		
	M2	~	~	~	~	~	~	1.1			
*	A1	~	~	~	~	~	~	1.2	1.1	1.3	1.4----->
	A2	~	~	~	~	~	~	1.1			
	S1	~	~	~	1.2						
	S2	~	~	~	1.2						
	B2	~	~	~	~	~	~				

average machine size-												
degree of confidence that method better than B1												
degree of confidence < .90 or method worse than B1												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
	M1	.97----->									.96	.90
	M2	.95----->								.96	.92	
	A1	.99----->									.98	
	A2											
	S1											
	S2											
	B2											

average machine size-%dec													
	maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
range	rank												
5.3 - 7.3	2	M1	7----->							7.3----->	7	5.3	
2.3 - 5.7	3	M2	5.3----->					5.4	5.6	5.7	4.9	3.8	2.3
7.8 - 8.9	1	A1	8.5----->					8.3	8.8----->	8.9	7.8		
1.1 - 1.9		A2	1.6----->		1.8----->		1.4	1.9	1.6	1.9	1.1		
		S1											
1.9 - 3		S2	2.1----->		2.2----->		1.9----->	2.4	3	2.9	2.1		
		B2	~	~	~	~	~	~	1.1	1.3----->	~		

Continued on next page.											

Appendix D continued

L2c	continued	Gav: speedup										Tav: speedup													
		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~	M1	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~
	M2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~	M2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~
*	A1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~	*	A1	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	A2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		A2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	S1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		S1	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	S2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		S2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	B2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		B2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
		Gmu: population-speedup										Tmu: population-speedup													
		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~	*	M1	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
	M2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	~		M2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
*	A1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~	*	A1	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	A2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		A2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	S1	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		S1	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	S2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		S2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
	B2	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4	~		B2	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	1.3	~
		probability of fewer generations-% inc										probability of less processor time-% inc													
		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	15	15	15	15	15	15	15	15	15	15	15	~	*	M1	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.4	9.4
	M2	15	15	15	15	15	15	15	15	15	15	15	~		M2	10	10	10	10	10	10	10	10	10	10
*	A1	17	17	17	17	17	17	17	17	17	17	17	~	*	A1	10	10	10	10	10	10	10	10	10	10
	A2	17	17	17	17	17	17	17	17	17	17	17	~		A2	10	10	10	10	10	10	10	10	10	10
	S1	17	17	17	17	17	17	17	17	17	17	17	~		S1	10	10	10	10	10	10	10	10	10	10
	S2	17	17	17	17	17	17	17	17	17	17	17	~		S2	10	10	10	10	10	10	10	10	10	10
	B2	17	17	17	17	17	17	17	17	17	17	17	~		B2	10	10	10	10	10	10	10	10	10	10

L3		worksheet																			
recommendations				S1, S2																	
average machine size																					
ranking		{M1, M2}, {A1, A2}																			
range		11.11 - 13.6																			
F=	0 - 35	lo		Gav=	8.54 - 42.29	lo															
	1 - 50	hi			10.79 - 63.37	hi															

~: same as B1
■ worse than B1
*: recommended
s: similar to B1
?: not conclusive

Failure rate & K-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
	M1	~	~	~	■	~	~	■	~	1.1	1.5	
	M2	~	~	~	1.1	~	■	■	~	~	1.2	
	A1	~	~	■	■	~	~	~	1.1----->	1.4		
s	A2	~	~	~	■	~	1.1	~	~	1.1	1.2	1.5
*	S1	~	~	~	~	1.2----->	1.1	1.2----->	1.4			
*	S2	~	~	~	1.2	1.3----->	1.2	1.3	1.4	1.3		
	B2	~	~	■	~	1.1	■	~	~	1.1	1.4	

average machine size-													degree of confidence that method better than B1														
degree of confidence < .90 or method worse than B1																											
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25		
M1		.999----->																									
M2		.999----->																									
A1		.99----->										.999-->		.99		.999											
A2		.99----->										.98		.97		.99											
S1												.90		.92		.93				.99							
S2												.91		.96----->		.97		.999									
B2																								.97			
average machine size-%dec																											
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25		
range	rank												range	rank													
9.3 - 14	1	M1	13----->										12	13	12	11	9.3	14									
12 - 14	1	M2	12----->										13	14	13	12	14										
5.6 - 11	2	A1	5.6----->		5.8	6.2	6.1	6.5	6.7	7.5	7.8	7.2	11														
5.5 - 7.7	2	A2	7.7----->		7.4	6.9----->		7.1	7.7	6.1	5.5	7.4															
1.8 - 7		S1	1.8----->		2.4----->		2.6	3	3.3	4.3	4.7	7															
1.5 - 9.4		S2	1.5----->		2	1.8	2.3	2.9	3.8----->		2.6	9.4															
1 - 5.1		B2	~	~	~	~	~	1.2----->		2.2	1.1	~	5.1														
														Continued on next page.													

Appendix D continued

L3		continued												
Gav-speedup														
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25		
*	M1	~	~	~	1.1----	>	1.2	1.3----	>	1.4----	>	1.1		
	M2					~			1.2	1.4	1.3	1.2----	>	
*	A1				1.2	1.4	1.3	1.2	1.4	1.3	1.2	1.4	1.1	
*	A2	1.1-----	>	1.2	1.5	1.3	1.2	1.3	1.4-----	>	1.2			
*	S1	1.2-----	>	1.4		1.2----	>	1.1	1.2	1.1	1.2	~		
*	S2	1.3-----	>	1.4	1.3	1.2-----	>	1.1-----			>			
*	B2	~	~	1.1	1.2	1.1	~	1.3----	>	1.2----	>			
Gmu:population-speedup														
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25		
s	M1	~	~	~	1.1----	>	~	~	1.1	1.2	1.3	1.5		
	M2									~	1.1	1.2		
	A1					~	~	1.1-----	>	1.2----	>	1.4		
*	A2	1.1-----	>	1.2-----			>	1.3	1.4	1.6				
*	S1	1.2-----	>	1.3-----			>	1.2	1.3	1.2	1.4			
*	S2	1.3-----	>	1.4-----			>	1.3	1.4----	>	1.3			
	B2					~	1.1----	>	~	1.1----	>	1.2	1.3	
probability of fewer generations-% inc														
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25		
*	M1	39-----					>	40	39	41	43	49	68	
*	M2	16-----			>	17-----		>	15	16	20	23	32	
*	A1	36-----			>	37-----		>	38	37	39	43	44	59
*	A2	49-----				>	50-----		>	51	52	57	76	
*	S1	30-----				>	31	30	31	30	33	32	43	
*	S2	44-----				>	45-----			>	47	48----	>	
*	B2	17-----				>	18----		>	17	18	20	22	30

Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		~	~	~	1.1-----	>	1.3	1.2	1.3	1.4	1.1	
M2					~			1.1	1.3	1.2	1.1-----	>
A1					1.1	~		~	~		1.1	
A2					1.1			~	~	1.1	~	
* S1		1.2-----	>	1.4	1.2	1.1----	>	1.2	1.1	1.2		
* S2		1.2-----	>	1.3----	>	1.1----	>	1.2	1.1	~	~	
s * B2		~	~	1.1----	>	~	~	1.2	1.3	1.2	1.1	~
Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s M1		~	~	~	~	~	~	~	1.1-----	>	1.2	1.5
M2									~	~	1.1	
A1										~	1.1	
A2										~	1.2	
* S1		1.2-----	>	1.3	1.2-----	>	1.3					
* S2		1.2-----	>	1.3-----	>							
B2				~	~	~	~	~	~	1.1	1.2	1.3
probability of less processor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		32-----	>		31	32	34	39	54			
* M2		9.1-----	>	9.2	9.3	8.7	7.6----	>	9.9	12	18	
* A1		3.5-----	>	3.6	3.5	2.8	3.2	4	4.1	9.9		
* A2		7.9-----	>	7.6	7.4	7.1	6.6	7.8	15			
* S1		24-----	>		23	25	24	33				
* S2		34-----	>		45	47	48----	>				
* B2		11-----	>	12----	>	11----	>	13	15	21		

Continued on next page.

Continued on next page.

Continued on next page.

Appendix D continued

L3c																									
continued																									
Gav-speedup													Tav:s-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1					~			1.2----->					M1								1.2----->				
M2									1.1----->				M2									~	~		
A1				1.1----->				1.1	~	1.3	~		A1										~		
A2	~	~		~			~	1.2	1.1	1.2	~		A2												
* S1	1.2----->	1.3	1.2	1.1	~	~	~	1.3	1.2----->	1.1			* S1	1.2----->	1.3	1.1----->	~	~	1.2----->	1.1	~				
s S2	1.1----->	~	~	~	~			~	1.1	1.3	~		S2	~	~	~	~				1.1	1.2			
s B2	~	~	~		~	~	1.1----->						B2					~	~	1.1----->					
Gmu:population-speedup													Tmu:population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1											1.1		M1											1.1	
M2													M2												
s A1	1.1	~	~		~	~	~	~	~	~	1.1		A1												
s A2	1.2----->	1.1	~	~	~	~			~	~	1.1		A2												
* S1	1.5	1.4----->					1.3----->	1.2----->					* S1	1.4----->	1.3----->	1.2	1.1	1.2----->							
* S2	1.3----->	1.2----->			1.1----->		~	~	~	1.2			s S2	1.2----->	1.1----->	~	~	~	~	~	~	~	1.1		
s B2	1.2	1.1----->		~	~	~	~	1.1----->					s B2	1.1----->	~	~	~	~	~	~	~	~	1.1		
probability of fewer generations-% inc													probability of less procesor time-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1											4.2		M1												
M2													M2												
* A1	7.3----->			7.2	7.5	7.9	7.1	8.8	7.1	14			A1												
* A2	3.1----->	3		3.1----->	2.4	1.1	2.4----->	7.1					A2												
* S1	31----->						30	32	3-3----->				* S1	23----->		22	23	21	22	23	22				
* S2	16----->				15	16	15----->	14	24				* S2	6.6----->	6.5----->	6.2	6.4	4.6	3.9	3.1	9.7				
B2													B2												

Appendix D continued

L4 worksheet											
recommendations				A2, S1, S2							
average machine size											
ranking		M2, A2, M1, A1									
range		11.89 - 13.07									
F=	0 - 17	lo	Gav=		7.49 - 21.42		lo				
	0 - 33	hi			10.85 - 42.08		hi				

~: same as B1											
worse than B1											
*: recommended											
s: similar to B1											
?: not conclusive											

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	~	~	~						~	1.1	~
M2	~	~	~	~					~	1.1	~
A1	~	~	~						~	1.3	1.2
?s	A2	~	~	~	~	~			1.1----->	1.3	1.6
s	S1	~	~	~	~	~	~		1.1	1.2	1.4
s	S2	~	~	~	~	~	~		1.2	1.1	1.2----->
s *	B2	~	~	~	~	~	~	1.2	1.1	1.2	1.1----->

average machine size-											
degree of confidence that method better than B1											
degree of confidence < .90 or method worse than B1											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	.99----->							.98	.99----->	.97	.98
M2	.999----->									.99----->	
A1	.96----->	.95----->			.94----->			.95			.95
A2	.99----->										
S1											
S2											
B2										.92	

average machine size-%dec											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
range	rank										
5.8 - 7.3	3	M1	7.3----->	7.1	7.2----->	6.9	6.5	6.9	7.3	5.8	6.8
6.8 - 8.4	1	M2	8.4----->		8.3	8.1----->	8.2	7.8	8	7.1	6.8
4.3 - 5.8	4	A1	5----->		4.5----->			4.7	5.2	4.3	5.8
7.1 - 7.9	2	A2	7.8----->			7.1----->			7.9	7.4	7.6
2.3 - 3.2		S1	2.6----->			2.3	2.8----->		3.2	2.5	2.9
1 - 2.1		S2	1.2----->				1.1----->		2.1	1.2	~
		B2	3.1----->				3	3.1	3.7	2.4	2.7

Continued on next page.											

Appendix D continued

L4	continued											
Gav-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1				1.1						~	~	
M2				~	1.1	~	1.1			~	~	
A1				1.2				1.5		1.3	~	~
* A2	1.3					1.5	1.4			1.3	~	
* S1	1.4					1.3	1.6		1.5	1.3	1.2	
* S2	1.3					1.2	1.4	1.2				
s B2	1.1		1.2	1.1		~	~	~		~	~	
Gmu:population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1								~	~	1.1		
M2						~	~	~	~	1.1	~	
A1				~	~	~	1.1	1.2	1.3	1.2		
* A2	1.3							1.4		1.5	1.4	
* S1	1.4								1.5			
* S2	1.3											
* B2	1.1										~	
probability of fewer generations-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1												
M2												
* A1	12								14	16	13	
* A2	28									30	32	
* S1	51								52	53	55	
* S2	33									34		
B2												

Tav:s-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1				~	~	1.1			~			
M2					~	~	~	1.1	~			
A1				1.1				1.5	1.4	1.3	~	~
* A2	1.2						1.4	1.3		1.2		
* S1	1.4						1.3	1.6	1.5	1.4	1.2	1.1
* S2	1.3						1.2	1.3	1.1			
?s B2	1.1						~	~	~			
Tmu:population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1									~	~		
M2									~	~		
A1							~	1.1	1.2	1.3	1.2	
* A2	1.2							1.3		1.4		
* S1	1.4								1.5			
* S2	1.3							1.2				
* B2	1.1							~	1.1	~	~	
probability of less procesor time-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1												
M2												
* A1	3.2								3.5	4.4	5.7	2.8
* A2	16										18	
* S1	38										40	
* S2	20									19		
B2												

L4c worksheet											
recommendations				A1, S1, S2							
average machine size											
ranking range				M2, M1, A2, S1							
F=	0 - 17	lo		Gav=	8.01 - 21.53	lo					
	0 - 42	hi			9.9 - 46.23	hi					

~: same as B1

worse than B1

*: recommended

s: similar to B1

?: not conclusive

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
s	M1	~	~	~	~	1.2	~	1.1----->	~		~
s *	M2	~	~	~	~	1.2	~	1.1----->	~	~	
s *	A1	~	~	~		~	~	1.4 1.5 1.3 1.2 1.1			
s *	A2	~	~	~		~	~	1.1 1.2 1.1----->			
*	S1	~	~	~	~	1.2----->	1.4 1.2 1.1 1.3 1.4				
s *	S2	~	~	~	~	~	1.1 1.4 1.3 1.2----->				
	B2	~	~	~			~	~			

average machine size-											
degree of confidence that method better than B1											
degree of confidence < .90 or method worse than B1											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	.98----->	.99----->							.98	.97	.99
M2	.999----->	.99----->							.99----->	.999	
A1											
A2	.99----->	.999									.999
S1	.93----->	.94	.95----->	.96----->	.97	.98					
S2											.97
B2											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
range	rank										
range	rank										
6.6 - 7.8	2	M1	6.6----->	6.9----->	7	7.3	7.4	7.1	6.8	7.8	
8 - 9.3	1	M2	8.4----->	8.6----->	8.5	8.2	8.1	8	9.3		
1.6 - 2.8		A1	1.6----->	2.1 2.2----->	2.5	2.8	2.7	1.9----->			
5.6 - 8.7	3	A2	5.6----->	5.7 5.8----->	5.9	6.4	6.7	7	8.7		
4.4 - 6.7	4	S1	4.4----->	4.6 4.7----->	5	5.3----->	5.7	6.7			
1.5 - 4.4		S2	1.5----->	1.7 1.8----->	2.2	2.4	3.1	4.4			
1.3 - 3.5		B2	1.5----->	1.4 1.3 1.7 1.9 2.1 3.5							

Continued on next page.

L4c		continued										
Gav-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
	M1	~	~	~			~			~	~	~
s *	M2	1.2----->			1.1	~	1.1	~	~	~	~	1.1
*	A1	1.3----->			1.6	1.5---->		1.1	~	1.1----->		
s *	A2	1.1----->			1.3	1.2---->		~	~	~	~	
*	S1	1.6----->			1.4	1.3---->		1.2	1.3	1.5	1.2	1.1
*	S2	1.1----->			1.2	1.5---->		1.4	1.2---->		1.1----->	
	B2											
Gmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s	M1	~	~	~	~	~	~	~	~			~
*	M2	1.2----->			1.1----->						~	~
*	A1	1.3----->			1.4----->				1.3	1.2---->		
*	A2	1.1----->						1.2	1.1----->		~	
*	S1	1.6----->			1.5----->		1.4----->					
*	S2	1.4----->			1.3---->		1.4----->				1.3	1.2
	B2											
probability of fewer generations-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
*	M1	3.4----->						3.6	3.9	2.8	2.5	9.2
*	M2	7.1----->						7	7.2---->		6.2	6
*	A1	27----->						28---->		27	24	26
*	A2	3.2----->						3.5	3.3	2.4	2	
*	S1	45----->								46		48
*	S2	40----->						41---->		40---->		
	B2											

Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
	M1	~	~	~	~		~			~	~	~
	M2	1.1----->					~					~
*	A1	1.3----->			1.5	1.4----	1.1			~	~	~
	A2	1.1----->			1.2	1.1----	~					
*	S1	1.6----->			1.4	1.2----	1.1	1.3	1.4	1.2	~	
*	S2	1.3----->			1.1	1.3----->	1.1----->			~	~	
	B2											
Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s	M1	~	~	~	~	~	~	~	~			
s *	M2	1.1----->					~	~	~	~	~	~
*	A1	1.3----->								1.2	1.1----->	
*	A2	1.1----->			~	1.1----->				~	~	~
*	S1	1.6----->			1.5----	1.4----->						
*	S2	1.3----->						1.4	1.3	1.2----		
	B2											
probability of less procesor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
	M1											
	M2											
*	A1	15----->								14	13	12
	A2											
*	S1	30----->								31	32	
*	S2	24----->								22	21	
	B2											

Appendix D continued

Appendix D continued

L5												
worksheet												
recommendations						NONE						
average machine size												
ranking			M2, A1, M1									
range			11.46 - 13.04									
F=	1 - 27	lo	Gav=			7.52 - 41.07	lo					
	5 - 39	hi				9.96 - 70.51	hi					

~: same as B1

worse than B1

*: recommended

s: similar to B1

?: not conclusive

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	~										
M2											
* A1	1.2----		1.1	1.2----		1.1----->					
A2	~				~						
S1	1.1----->										
S2	~	1.1		~	~		~	~	~		
* B2	~	~	1.5	1.1----->		1.2	~				

average machine size-											
degree of confidence that method better than B1											
degree of confidence < .90 or method worse than B1											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	0.98----->					.97----->		.96	.97	.93	.91
M2	.999----->		.99	.999----->		.99----->				.97----->	
A1	.95	.94----->	.93	.92	.91----->		.90	.93			
A2											
S1											
S2											
B2											

average machine size-%dec												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
range	rank											
3.7 - 5	3	M1	4.9----->		5	4.7----->		4.4	4.8	3.7	3.8	
5.5 - 8.7	1	M2	8.6----->		8.4	8.7	8.6	8.5	8.3	7.1	6.8	
4.5 - 5.2	2	A1	5.2	4.9	5	4.9----->		4.6	4.5----->	5.1	4.6	
		A2	1.9	1.8	1.5----->		1.7	1.8	~	~	3.2	
		S1										
		S2										
		B2										

Continued on next page.

Appendix D continued

L5												
continued												
Gav-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
s	M1	~	1.2	~	~	~	1.2	~	1.1	~	~	1.3
	M2		~						~	1.1----	>	
s *	A1	1.5	1.8	1.1	~	1.1----	>	~	1.1	~	~	
	A2	1.1	1.3	~		~			~	~	1.2	
*	S1			~	1.2-----			1.3	1.2	1.1----	>	
s	S2	~	1.2		~	1.1	~	~	1.1----	>	~	1.2
	B2	1.2-----	>						~		1.1	
Gmu:population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
	M1						~	~	~	~	~	
	M2											
*	A1	1.1	1.2	1.3----	>	1.2-----				1.1----	>	
	A2										~	
	S1						~	~	1.1-----	>		
s	S2	~	~	~	~	~	~	~	~	~	~	
	B2	1.2-----	>	~	~	~	~					
probability of fewer generations-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
*	M1	8.8-----	>	9.4	9.6	9.5	9.2	10-----	>	11	10	9.5
	M2											
*	A1	3.2-----	>	3.4	3.5	3.3	3.5----	>	3	3.2	1.7	2.5
*	A2	1.5-----	>	1.9	2.1	2	1.8	2.3	2.8	1.4	2.5	3
*	S1	13-----						14	15	16	17	19
*	S2	9.7-----	>	10	9.8	9.7	9.9----	>	10	9.7	11	8.1
	B2											

Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s	M1	~	1.2	~	~	~	1.2	~	1.1	~	~	1.2
	M2										~	~
s	A1	1.4	1.7	~	~	~	~	~	~	~	~	
	A2	~	1.2	~						~		1.1
*	S1		1.1	~	1.1----->		1.2----->		1.3	1.2	~	~
?s	S2				~	~	~	~	~	~	~	1.1
	B2	1.2----->										~
Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
	M1									~	~	~
	M2											
*	A1	1.1----->	1.2----->						1.1----->		~	
	A2											
	S1								~	~	1.1----->	
	S2			~			~	~	~	~	~	
	B2	1.1----->	~	~	~	~						
probability of less procesor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s	M1	~	~	~	~	~	~	~	~	~	~	
	M2											
	A1											
	A2											
*	S1	4.2----->					4.3	4.5	4.8	5.7	7	7.6
	S2	~	~	~	~		~	~				
	B2											

L5c	worksheet														
recommendations	NONE														
average machine size															
ranking	M2, M1														
range	11.23 - 12.72														
F=	0 - 29	lo	Gav=	7.66 - 47.52	lo										
	2 - 42	hi		9.72 - 72	hi										

Appendix D continued

L5c																									
continued																									
Gav: speedup													Tav: s-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	~	~	~						~	1.1			M1									~	~		
M2													M2												
?s A1			1.1	1.2----	>	~	~	~	1.2	1.1	~		?s A1			~	1.2	1.1	~	~	1.1-----	>			
A2									1.2----	>	1.1		A2									1.1	~		
S1									1.2	~	1.3	1.1	S1								1.2	~	1.2	1.1	
S2								~	1.1-----	>	1.3	~	S2								~	~	1.2		
B2	1.1-----	>		~				~	~	~			B2	~	~						~	~			
Gmu: population-speedup													Tmu: population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1											~		M1												
M2													M2												
A1								~	~	~	1.2		A1									~	1.1		
A2													A2												
S1											~		S1										~		
S2											1.1		S2										~		
B2											~		B2										~		
probability of fewer generations-% inc													probability of less procesor time-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1													M1												
M2													M2												
* A1	9.7-----	>	9.8	9.9	10	11-----	>	18					s A1	~	~	~	~	~	~	~	~	~	1.2	6.4	
A2													A2												
* S1	4.1-----	>	4	4.3----	>	4.2	3.8	5.5	5.3	12			S1										~		
* S2	1.3	1.4	1.3----	>	1.5	1.8	1.2	~	1.9	2	9.4		S2												
B2													B2												

Continued on next page.

Appendix D continued

L6																									
continued																									
Gav-speedup													Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		1.1----->			1.3	1.5----		1.3----		1.2	~		* M1		1.1----->			1.2	1.4----		1.2	1.3	1.1	~	
* M2		1.1----->						1.3	1.4	1.3	1.2	~	s * M2		~	~	~	~	~	~	1.2	1.3	1.2	1.1	
* A1		1.6----->						1.4----		1.3	1.2	1.1	* A1		1.5----->						1.3----		1.2	1.1	~
A2			~	1.2----		1.3	1.1	1.2	1.1	~			A2			~	1.1----		1.2	~	1.1	~			
?* S1				~	1.2	1.3	1.2	1.3	1.2	~	~		S1				~	1.1	1.2	1.1	1.2	1.1	~		
S2			~	~	~	1.2	~	1.2	1.1	~	~		S2						1.1		1.1	~			
B2						~	~	1.1					B2						~	~	~				
Gmu:population-speedup													Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		1.1----->					1.2----		1.3----->		1.2		* M1		1.1----->				1.2----->				1.3	1.2	
* M2		1.1----->								1.2----		1.3	s M2		~	~	~	~	~	~	~	~	1.1	1.2----	
* A1		1.5----->										1.4	* A1		1.5----->						1.4----->				
A2					~	~	~	~	1.1----->				A2									~	~	~	
S1						~	~	~	1.1	1.2	1.1		S1							~	~	1.1----->			
S2							~	~	~	1.1----->			S2									~	~	~	
B2						~	~	~	~	~	~	~	B2									~	~		
probability of fewer generations-% inc													probability of less procesor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		15----->								16	17	15	* M1		6.6----->							6.8	7.5	5.6	
* M2		28----->								29	30	33	* M2		16----->				15----		16----			18	
* A1		60----->								61	62		* A1		41----->									42	
A2													A2												
* S1		11----->								13	12		* S1		1.8----->						1.7	2.2	3.2	2.4	
* S2		6.9----->				7	6.9	7.7	9.2	8.1			S2												
B2													B2												

Appendix D continued

L6c worksheet recommendationsM1, A1, A2, S1 average machine size rankingA2, M1, M2 range11.38 - 13.06 F=0 - 19loGav=8.96 - 21.74lo 0 - 32hi10.14 - 44.87hi ~: same as B1 worse than B1 *: recommended s: similar to B1 ?: not conclusive Failure rate & K-speedup maxgen-->1000750500300250200150100755025 *M1~ ~ ~1.2---->1.31.41.3~ ~1.1 M2~ ~ ~ ~ ~1.1 *A1~ ~ ~1.2---->1.31.21.41.11.21.3 ?*A2~ ~ ~1.2---->1.3 ~1.11.11.3 ?*S1~ ~ ~ ~1.1----->1.21.5 S2~ ~ ~ ~ ~ ~ ~ ~1.3 *B2~ ~ ~1.2---->1.3 ~1.3 ~1.1----->											
average machine size- degree of confidence that method better than B1 degree of confidence < .90 or method worse than B1 maxgen-->1000750500300250200150100755025 M1.99----->.97----->.93 M2.96----->.95.94.96.95----->.94.92.90 A1 A2.98----->.97----->.96.95.94.93 S1 S2 B2 average machine size-%dec maxgen-->1000750500300250200150100755025 range rank 1.6 - 7.82M17.8----->7.7----->7.36.55.64.91.6 1.7 - 5.83M25.75.4---->5.25.8---->5.75.34.74.61.7 1.1 - 5.3A15.3----->5.2----->4.93.83.62.51.1 4.8 - 7.41A27.4----->7.1----->6.86.76.54.8 S1~ ~ ~ S22.32.4---->2.5---->2.42.3~1.6 B2 Continued on next page.											

Appendix D continued

L6c		continued											
Gav-speedup													
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	
*	M1	1.4----->			1.1----->		~	~		1.1----->		~	
	M2					~	1.3	1.2	~	1.3	1.2	~	
*	A1	1.6----->			1.2----->		1.1	1.2	~	1.1	~	~	
*	A2	1.2----->			1.1----->		~	1.2	~	1.3	1.1		
*	S1	1.1----->		1.4	1.5----->		1.3	1.2	1.3----->		1.2	~	
	S2	1.1					~		1.2	1.3	1.1		
	B2	1.2----->			~	~	~	1.1		~			
Gmu:population-speedup													
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	
*	M1	1.4----->				1.3----->		1.2	1.1----->				
	M2											~	~
*	A1	1.5----->				1.4----->		1.3----->		1.2----->		1.3	
*	A2	1.2----->				1.1----->							1.2
*	S1	1.1----->			1.2----->			1.3----->					1.4
	S2							~	~	~	1.1	1.2	
*	B2	1.2----->			1.1----->						~	~	~
probability of fewer generations-% inc													
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25	
*	M1	16----->								15----->			
*	M2	10----->							11	10	12	17	
*	A1	23----->								22	23	27	
*	A2	15----->								16			18
*	S1	38----->						39----->			40	45	
*	S2	11----->									12	17	
	B2												

Tav:s-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
s M1		1.4----->			1.1----->		~	~		1.1----->		~
	M2					~	1.2	~	~	1.2	1.1	
* A1		1.5----->	1.4	1.1----->		~	1.1	~	1.1	~		
s A2		1.1----->			~	~		1.1	~	1.2	~	
* S1		1.1----->	1.3	1.4----->	1.3	1.2----->				1.1		~
	S2		~			~		1.1	1.2	1.1		
	B2	1.2----->		~	~		~		~			
Tmu:population-speedup												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		1.4----->			1.3----->		1.2----->		1.1	~	~	~
	M2											~
* A1		1.5----->			1.4----->		1.3	1.2----->	1.1	1.2----->		
s A2		1.1----->				~	~	~	~	1.1----->		
* S1		1.1----->			1.2----->					1.3----->		
	S2										~	1.1
s * B2		1.2----->		1.1----->		~	~	~	~	~	~	~
probability of less procesor time-% inc												
maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
* M1		7.1----->						7	6.6	5.9----->		5.8
* M2		1.3----->		1.2	1.1	~	1.1	1.7	~	1.9		5.1
* A1		13----->						12----->				15
* A2		5.6----->						5.2	5.4	4.7	5.5	5.9
* S1		27----->						28----->				32
* S2		1.8----->		1.7----->				1.4	1.1	2.4	5.5	
	B2											

Appendix D continued

L7 worksheet											
recommendations				NONE							
average machine size											
ranking		M2, M1, A1									
range		11.31 - 12.97									
F=	0 - 24	lo	Gav=	8.12 - 39.4	lo						
	3 - 39	hi		10.67 - 65.88	hi						

~: same as B1

worse than B1

*: recommended

s: similar to B1

?: not conclusive

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	1.2	~						1.1----->	1.2	1.4	
M2	1.2	~					1.1----->	~	1.1----->		
A1						~	1.1	~	1.1	1.2	
*	A2	1.2----->	~	1.2	1.1----->	1.2	1.1		1.1	1.2	
	S1	1.2	~								
s	S2	1.2----->	~	1.1----->	~	~	1.1	~	1.1	1.2	
	B2	~	~								

average machine size-												
degree of confidence that method better than B1												
degree of confidence < .90 or method worse than B1												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	.999	.99----->								.98	.97	
M2	.999----->											
A1	.98----->	.97	.98----->					.99	.98	.96		
A2							.90----->	.91				
S1								.92	.93----->			
S2												
B2												

average machine size-%dec												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
range	rank											
5.3 - 7.8	2	M1	7.8	7.5	7.2	6.9	6.1----->	6.2	6.8	6.9	5.8 5.3	
11 - 13	1	M2	11----->						12----->	11	13	
4.9 - 6.1	3	A1	5.8	5.6	5	5.2----->	5.3	5.5	5.9	6.1	5.6 4.9	
2.9 - 3.5		A2	3.3----->	3.1	3.3	3.2	3.3	3.4	3.5	3	2.9 3.3	
2.2 - 4.2		S1	2.8	2.6	2.4	2.2	2.5	2.4	2.8	3.8	3.9 4.2 3.3	
1.3 - 2.2		S2	2.1	1.9	2----->	1.7	1.6	1.3	2.2----->	2.1		
		B2								~	~	~ ~

Continued on next page.

Continued on next page.

Appendix D continued

L7											
continued											
Gav-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
* M1			~	1.2	1.7----		1.6	1.2	1.3----		~
* M2			1.4	1.2-----			~	~	1.2	~	1.1
* A1			1.3	1.1-----	1.4	1.2	~	1.2	1.1-----		
s A2	~	~	1.2	~	~	1.1		~	1.2	1.1	~
S1						~	~	1.1-----			
s * S2			1.1	~	~	1.1	1.2	~	1.1-----		~
B2								1.1	~		
Gmu:population-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1				~	~	1.1-----		1.3-----			1.4
M2				~	~	1.1-----					
A1						~	~	1.1-----			1.2
* A2	1.2-----	1.1	1.2	1.1-----				~	1.1	1.2	
S1									~	~	
* S2	1.2	1.1	~	1.1-----	~	~	1.1-----				1.2
B2											
probability of fewer generations-% inc											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
* M1	30-----						31	33	32	34	39
* M2	12-----			13-----					12	15-----	
* A1	22-----			23-----	22	23	24	23	27	33	
* A2	18-----								17	20	25
s * S1	~	~	~	~	~	~	1.2	1.5	1.2	3.2	4.2
* S2	13-----							14	13	15	18
B2											

Tav:s-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1				1.2	1.6----		1.5	1.2	1.3	1.2	~
M2			1.3	1.1-----			~	1.1	~	~	~
A1			1.2	1.1-----	1.3	1.1	~	1.1	~	1.1	
A2			1.1					1.1	~	~	~
S1						~	~	1.1	~	1.1	
S2						~	1.1		~	~	~
B2								1.1	~		
Tmu:population-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1					~	~	1.1	1.2-----		1.3-----	
M2				~	~	~	~	~	~	1.1-----	
A1							~	~	~	1.1	1.2
s A2	1.1-----	~	1.1	~	~	~	~	~	~	1.1-----	
S1											
s S2	1.1	~	~	~	~	~	~	~	~	~	1.1
B2											
probability of less procesor time-% inc											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
* M1	18-----						17-----	18	19-----		24
* M2	1.8	1.7	1.5	1.6	1.5	1.6	1.8	1.5	~	2.1	~
* A1	11-----			12-----			11	12-----	11	13	17
* A2	6.6-----			6.5-----			6.3	6.2-----	5.7	4.4	6.4
S1											
* S2	2.2	2.1	2-----		1.9	1.8	1.2	2	1.1	1.6	3.1
B2											

Appendix D continued

L7c worksheet												
recommendations						M2						
average machine size												
ranking		M1, M2, A2, A1										
range		11.83 - 13.15										
F=	0 - 24	lo	Gav=		7.44 - 41.34		lo					
	2 - 34	hi			10.78 - 96.14		hi					

average machine size-												
degree of confidence that method better than B1												
degree of confidence < .90 or method worse than B1												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	.999----->										.99	
M2	.999	.99	----->					.999	.99	----->		.96
A1	.99	----->						.97	.95	.93		
A2	.99	----->									.97	
S1												
S2												
B2												

~: same as B1											
worse than B1											
*: recommended											
s: similar to B1											
?: not conclusive											

Failure rate & K-speedup													
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		
M1	1.2	1.4									~		
*	M2	1.2----->	1.3	1.1----->	~	1.1	~	1.2	1.3				
	A1	~	1.1										
*	A2	1.2----->	1.3	1.2----->	1.3	1.1	~	1.1	~				
	S1	~	1.1	~								1.1	
s	S2	~	1.2----->	1.1----->	~	~	~	~	~				
	B2	1.2----->	1.3	1.1								~	~

average machine size-%dec											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
range	rank										
7.8 - 10	1	M1	10	9.7---->	9.5---->	9.7	8.8	8.7	8.2	9	7.8
5.6 - 8.7	2	M2	8.7	7.8	7.9----->	8.2	8.4	7.9	7.3	7.5	5.6
3.5 - 6.7	4	A1	6.7	6.3---->	5.9----->	5.8	5.1	4.7	4.3	3.5	
1.9 - 6.8	3	A2	6.8	6.4	6.5	6.4---->	6.2	6.5	6.2	5	1.9
1.8 - 3		S1	1.9	1.8	2.1	2.5---->	2.6	2.7	3	2.8	2.2
1.9 - 2.7		S2	2.7	2.3	2.4	2.5---->	2.4---->	2.6	2.1	1.9	2.2
1 - 1.7		B2	1.7	1.3	1.4	1.2	~	~	~	~	~

Continued on next page.											

Continued on next page.

average machine size-

	degree of confidence that method better than B1										
	degree of confidence < .90 or method worse than B1										
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	.999----->			.99----->					.98----->		
M2	.999----->						.99	.98			.97
A1	.99----->								.97	.95	
A2	.999----->						.99	.98	.90		
S1	.93----->					.91----->		.90			
S2	.94	.92				.91				.90	
B2											

average machine size-%dec													
	maxgen-->		1000	750	500	300	250	200	150	100	75	50	25
range	rank												
3.1 - 8	1	M1	7.5	7.3	6.4	6.2	6.7	6.8	7.6	8	7.5	7.3	3.1
3.4 - 8.8	1	M2	8.8	8.5	7.3	7.1	7.4	7.6	6.9	6.8	4.1	8.4	3.4
1 - 7.5	2	A1	6.8----->		5.8	6.8	5.9	6.8	6.6	7.5	6.4	5.9	~
1.6 - 7.7	2	A2	5.8	6	6.4	7.7	7.6	7.3	7	5.9	4.5	5.8	1.6
		S1	3	3.3	2.6	3.1----->		2.9---->		1.9	~	~	
		S2	2.5	2.2	1.8	2.2	1.7	1.8	2.4	2.6	2.2	4.3	
		B2	~	~	~	~	~	~	~	~	1.5	2.2	

Appendix D continued

L8	continued																							
Gav-speedup												Tav:s-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1					~		~	~	~	~		M1								~	~	~		
M2												M2												
s A1	1.4	1.3	~	~	1.1	----->	~	~	~	~		A1	1.1	----->										
s* A2	1.6	1.5	1.2	1.1	----->	~	~	~	~	~		A2	1.2	1.1	~					~	~	~		
s S1	1.3	1.2	~	~	1.1	----->	~	~	~	~		S1	1.2	1.1	~	~	~	1.1				~	~	
S2	1.3	----->	1.1	~	~	1.1	~	~	~	~		S2	1.2	----->	~		~	1.1				~	~	
?s B2	1.2	1.4	1.2	~	1.1	~	~	~	~	~		B2	1.2	1.3	1.2	~	~		~	~		~	~	
Gmu:population-speedup												Tmu:population-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	~											M1												
M2	1.1	~	~	~								M2	~											
* A1	1.7	----->	1.5	1.4	----->	1.3	1.4	----->	~	1.4	----->	* A1	1.4	1.3	----->	1.2	1.1	----->	~	1.1	----->	~	1.1	----->
* A2	1.8	----->	1.7	1.6	----->	1.7	1.6	1.7	----->	1.6		* A2	1.4	----->	1.3	----->	1.3	----->	1.2	1.3	----->	1.2	1.3	----->
* S1	2	1.9	1.7	1.6	1.5	----->	1.6	1.4	----->	1.3	1.4	* S1	1.9	1.8	1.7	1.5	----->	1.4	1.5	1.4	----->	1.2	1.4	----->
* S2	1.8	1.7	----->	1.6	1.5	1.4	1.6	1.3	1.5	1.4	1.2	* S2	1.7	1.6	----->	1.5	1.4	1.3	1.5	1.3	1.5	1.3	1.1	----->
* B2	1.3	----->	1.4	----->	1.3	1.4	----->	1.3	----->	1.4	~	* B2	1.3	----->	1.4	1.3	----->	1.4	1.3	1.2	1.3	1.3	~	----->
probability of fewer generations-% inc												probability of less processor time-% inc												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1												M1												
M2												M2												
* A1	43	----->	45	32	37	----->	39	32	41	40	44	* A1	19	----->	14	10	----->	11	9.8	12	11	13		
* A2	59	60	62	----->	59	60	63	58	68	72	60	* A2	30	----->	31	29	26	25	28	22	29	32	21	
* S1	49	----->	48	45	44	42	48	41	38	27	45	* S1	46	----->	45	41	40	38	43	37	33	22	39	
* S2	47	46	47	45	42	40	50	34	50	36	20	* S2	40	----->	37	34	31	40	26	40	28	28	9.8	
* B2	28	27	29	32	30	31	34	28	26	34		* B2	25	24	26	28	26	28	31	29	22	29		

Appendix D continued

L8c											
worksheet											
recommendations A1, A2, S1, S2											
all fail tests for Gav & Tav's when failure rate high											
average machine size											
ranking M1, M2, {A2, A1}, {S1, S2, B2}											
range 12.09 - 14.07											
F= 2 - 85 lo Gav= 14.13 - 156.2 lo											
13 - 93 hi 17 - 271.49 hi											
~: same as B1											
worse than B1											
*: recommended											
s: similar to B1											
?: not conclusive											
Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1											1.1
M2											
* A1	~	~	1.1	1.3----->	1.2	1.1----->			~		1.2
* A2			~	1.1	1.2	1.1	1.2	1.4	1.3----->		1.2
* S1	1.1----->		1.4	1.6	1.7	1.5	1.6	1.5	1.4----->		
* S2	1.1	~	1.1	1.3	1.4	1.3	1.5	1.7----->			1.1
? B2	1.4	1.2	1.1----->			1.1		~			

average machine size-											
degree of confidence that method better than B1											
# : insufficient number of data points											
degree of confidence < .90 or method worse than B1											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1	.999----->									.99	.96
M2	.999----->									.98	.95 #
A1	.999----->	.99----->								.98	.91
A2	.999----->	.99	.999----->							.99	
S1	.99----->		.999	.98	.97	.91					
S2	.97	.98	.99----->	.98	.97	.99	.98	.97			
B2		.91	.94	.96----->	.95	.98----->	.99	.98	.97		
average machine size-%dec											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
range	rank										
8.1 - 10	2	M1	8.6	8.1	9.3	8.9	9.2	9.8----->	9	10----->	8.7
2 - 12	1	M2	10	11----->				12	11----->	8	7.7 2
3.4 - 7.6	3	A1	7	7.4	7.6	7.3	6.2	6	6.4	6.2	6.1 3.4 4.2
4.2 - 9	3	A2	6.1	6.2	6.6----->	5.6	6.3	7.9	8.3	9	7.1 4.2
-6.6	4	S1	4.9	5.1	5.6	5.8----->	6.3	6.6	4.9	5	4.3
1 - 6.5	4	S2	3.5	4.1	5.1	5.5	4.7----->	6.3	6.5	6.2	4.4 ~
1.1 - 10	4	B2	1.1	2.1	2.6	3.2	3.9	3.5	4.7	5.9	8.1 7.1 10
Continued on next page.											

Appendix D continued

L8c																										
continued																										
Gav-speedup													Tav:s-speedup													
maxgen-->													maxgen-->													
*	M1	1.1	~	1.1----->	1.2	1.1----->	~	~	~				?s	M1	~	~	1.1	~	1.1----->							
	M2													M2												
?s	A1	1.1	1.2----->	~		~	~	~						A1	1.1----->	~								~		
?	A2	1.1	1.2----->	1.1----->			1.1	~						?s	A2	1.1	~	1.1	~	~	1.1	~				
	S1	1.4	1.5	1.3	1.1	~	1.1							S1	1.4----->	1.2	1.1	~	~							
*	S2	1.3	1.4----->	1.2	1.1	1.3	1.1	~						?	S2	1.2	1.3----->	1.2	~	1.2	1.1			~		
	B2	~	~	~	~	~								B2												
Gmu:population-speedup													Tmu:population-speedup													
maxgen-->													maxgen-->													
	M1													M1												
	M2													M2												
*	A1	1.1----->	1.2----->	1.3	1.2	1.1----->	~	1.2						*	A1	1.1----->	1.2----->	1.1----->	~	1.1				1.1		
*	A2	~	~	1.1	1.2----->		1.4	1.3	1.2----->					*	A2	~	~	~	1.1----->	1.3	1.2	1.1----->				
*	S1	1.4----->	1.5	1.6----->	1.5----->	1.4----->	1.3							*	S1	1.4----->	1.5----->	1.6	1.4	1.5	1.4	1.3----->				
*	S2	1.2----->	1.3	1.4	1.5	1.4	1.5	1.7----->	1.6	1.1				*	S2	1.2----->	1.3	1.4	1.3	1.4	1.6----->	1.5	~			
	B2	1.1----->		~	~	~								B2	~	~	~	~		~						
probability of fewer generations-% inc													probability of less procesor time-% inc													
maxgen-->													maxgen-->													
	M1													M1												
	M2													M2												
*	A1	14----->	15	18	19	15	11	10	11	13	19			*	A1	9.1----->	9.7	12	13	8.6	5	4.2	4.5		11	
*	A2	16----->	17	20	21	20	23	32	26	22	16			*	A2	8.3	8	9.2	11	12	10	13	21	15	12	6
*	S1	40	41	42	44	46	40	43	36	30	25			*	S1	37----->	38	40	41	36	39	31	26	21		
*	S2	37----->	39	42	46	43	49	59----->	54	10				*	S2	31	30	32	34	38	35	41	49	50	45	2.1
	B2	1.8	1.6	~										B2												

Continued on next page.

L9	worksheet
recommendations	B1
average machine size	
ranking	{M1, M2}, A2, A1
range	11.04 - 13.19
F=	0 - 43 lo Gav= 8.94 - 61.19 lo 5 - 67 hi 12.04 - 103.90 hi
~: same as B1	
worse than B1	
*: recommended	
s: similar to B1	
? : not conclusive	
Failure rate & K-speedup	
maxgen-->	1000 750 500 300 250 200 150 100 75 50 25
M1	~ ~
M2	~
A1	~
A2	
S1	
S2	~
B2	~ ~

L9													continued												
Gav-speedup																									
maxgen-->																									
	M1																								~
	M2																								1.1
	A1			~																			~		~
	A2					~	~	~																	
?	S1					1.1----	>	1.2	1.1	~	~	1.1													1.1
?	S2			~	~	1.1	~	~	1.1-----	>															~
	B2									~															~
Gmu:population-speedup																									
maxgen-->																									
	M1																								
	M2																								
	A1																								
	A2																								
	S1																								
	S2																								
	B2																								
probability of fewer generations-% inc																									
maxgen-->																									
	M1																								
	M2																								
	A1																								
	A2																								
	S1																								
	S2																								
	B2																								
Tav:s-speedup																									
maxgen-->																									
	M1																								~
	M2																								1.1
	A1																								
	A2																								
	S1																								1.1
	S2																								
	B2																								~
Tmu:population-speedup																									
maxgen-->																									
	M1																								
	M2																								
	A1																								
	A2																								
	S1																								
	S2																								
	B2																								
probability of less procesor time-% inc																									
maxgen-->																									
	M1																								
	M2																								
	A1																								
	A2																								
	S1																								
	S2																								
	B2																								

Appendix D continued

Appendix D continued

L9c worksheet recommendations B1 average machine size ranking M1 range 11.23 - 13.12 F= 0 - 45 lo Gav= 9.64 - 59.16 lo 5 - 58 hi 12.29 - 117.05 hi												
~: same as B1 worse than B1 *: recommended s: similar to B1 ?: not conclusive												
Failure rate & K-speedup												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	~											
M2											~	
A1	1.2	1.1	~									
A2												
S1	~									~	1.1	
S2												
?	B2	1.2	1.3	1.4----->	1.1----->							

average machine size-												
degree of confidence that method better than B1												
degree of confidence < .90 or method worse than B1												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
M1	.99	.98----->			.97----->		.96----->		.92			
M2												
A1												
A2												
S1												
S2												
B2												

average machine size-%dec												
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25	
range	rank											
1.7 - 6.8	1	M1	6.8	6.3	6.2	6.1	5.2	5.5	5.7	5.4	4.4	3.9
		M2	3.6	3.2	2.9	2.2	2.1	2	1.2	1.5	~	~
		A1	1.1----->	1.2	~	~	~	~	~			
		A2	~	~	~	~	~	~	~			
		S1										
		S2										
		B2										

Continued on next page.

Continued on next page.

L10													continued												
Gav-speedup													Tav:s-speedup												
maxgen--> 1000 750 500 300 250 200 150 100 75 50 25													maxgen--> 1000 750 500 300 250 200 150 100 75 50 25												
M1 ~ 1.2 1.1 1.4----> 1.2													M1 1.2 1.1 1.3 1.4 1.2												
M2 ~ 1.2													M2 1.1												
s * A1 1.1 ~ 1.2 ~ ~ ~ 1.1-----> 1.2 1.1													A1												
A2 ~ ~ ~ ~ 1.3 1.1-----> 1.2 1.1													A2												
S1 1.4 1.3 1.1 ~ ~ 1.1 1.2 1.1 1.2 1.1 1.2													* S1 1.3-----> 1.1 ~ ~ 1.1 1.2 1.1----->												
* S2 ~ ~ ~ ~ ~ ~ ~ ~ 1.1													S2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~												
B2 1.2 1.1 ~ ~ ~ ~ 1.1 1.3 1.2 ~													B2 1.2 ~ ~ ~ ~ 1.1 1.2-----> ~												
Gmu:population-speedup													Tmu:population-speedup												
maxgen--> 1000 750 500 300 250 200 150 100 75 50 25													maxgen--> 1000 750 500 300 250 200 150 100 75 50 25												
M1													M1 1.1												
M2													M2												
s * A1 ~ 1.1 ~ 1.1-----> ~ 1.1----->													A1												
?s A2 ~ ~ 1.1 ~ ~ ~ ~ ~ 1.1													A2												
* S1 ~ ~ ~ 1.2 1.1-----> 1.3 1.2 1.4 1.3													* S1 ~ 1.1-----> ~ ~ 1.2-----> 1.3 1.2												
S2 ~ ~ ~ ~ ~ ~ ~ ~ ~ ~													S2												
B2 1.3-----> 1.1 ~ ~ ~ ~ 1.1													B2 1.3-----> 1.1 ~ ~ ~ ~ 1.1												
probability of fewer generations-% inc													probability of less processor time-% inc												
maxgen--> 1000 750 500 300 250 200 150 100 75 50 25													maxgen--> 1000 750 500 300 250 200 150 100 75 50 25												
M1 20													M1 14												
M2													M2												
* A1 7.8 8.1 7.2 8.9 9-----> 7.7 15 13-----> 25													A1												
A2 1.2-----> 2.1 ~ ~ ~ ~ 3.2 3.9 10 17													A2												
* S1 16 17 19 20 19 17-----> 30 27 41 34													* S1 13 14 16 17 15 13-----> 24 22 35 28												
S2													S2												
B2 2.2 2.4 2.6 ~ ~ ~ ~ 16													B2 9.5												

Appendix D continued

L10c worksheet												
recommendations NONE.												
S1 good for maxgen>=200 and for criteria using F												
average machine size												
ranking		M1, M2, A1, A2, S1										
range		12.68 - 14.27										
F=	2 - 78	lo	Gav=		12.5 - 145.05		lo					
	21 - 93	hi			16.18 - 240.38		hi					

average machine size-													
degree of confidence that method better than B1													
# : insufficient number of data points													
degree of confidence < .90 or method worse than B1													
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		
M1	.999----->										.99	#	
M2	.999----->		.99	.999		.99----->		.96	.92	.97	.95		
A1	.999----->										.99	.98	.97
A2	.99----->		.999---		.99----->		.98	.96	.97	.94			
S1	.98----->		.97----->		.99----->		.98	.96	.95	.96	.97		
S2							.90	.91	.94				
B2									.94				

~: same as B1											
worse than B1											
*: recommended											
s: similar to B1											
?: not conclusive											

Failure rate & K-speedup											
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25
M1											
M2											
A1	1.2	~	1.1----->		1.2	1.1	~				
A2	1.3----->		1.2----->		~	1.1					
*	S1	1.8----->		1.6	1.2	1.3	1.2----->		1.1----->		~
	S2	1.3----->		1.1	~	1.2	~				
	B2	1.3	1.2----->		1.3----->		1.1				

average machine size-%dec													
maxgen-->	1000	750	500	300	250	200	150	100	75	50	25		
range	rank												
8.1 - 9.4	1	M1	9.4	8.9	8.6	8.8	9.1	8.5	9.2	8.8	8.9	8.1	8.3
4.7 - 8.7	2	M2	8.7	8.6	8.5	7	7.4----->		7.6	5.4	4.7	6.6	7.9
3.6 - 7.2	3	A1	6.9	6.8	6.9	7.2----->		6.2	6.4	5.6	5.1	4.7	3.6
4.4 - 6.2	4	A2	5.1----->		5.9	6.2	5.5	5.1	4.9	4.5	4.4----->		
2.9 - 5.7	5	S1	3.1----->		2.9	3.1	3.8	4----->		3.3----->		4	5.7
1.3 - 4.2		S2	1.3	1.4----->		1.8	2.5	2.9	3	2.6	1.8	4.2	
1.2 - 2.8		B2	1.5	1.6	1.3	1.6	2	1.9	1.5	2.8	1.9	1.2	1.9

Continued on next page.

Continued on next page.

[illegible][illegible]

References

- Adami, Chris, and C. Titus Brown. (1994) "Evolutionary Learning in the 2D Artificial Life System 'Avida'." In *Artificial Life IV*, pp. 377-381, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Angeline, Peter J. (1996) E-mail received Jan. 7, 1996.
- Angeline, Peter J. (1994) "An Alternate Interpretation of the Iterated Prisoner's Dilemma and the Evolution of Non-Mutual Cooperation." In *Artificial Life IV*, pp. 353-358, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Angeline, Peter J., and Jordan B. Pollack. (1994) "Coevolving High-Level Representations." In *Artificial Life III*, pp. 55-71, edited by Christopher G. Langton. Reading, MA: Addison-Wesley.
- Angeline, Peter J., and Jordan B. Pollack. (1993) "Evolutionary Module Acquisition." In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pp. 154-163, edited by D.B. Fogel and W. Atmar. Palo Alto, CA: Morgan Kaufmann.
- Axelrod, R. (1987) "The Evolution of Strategies in the Iterated Prisoner's Dilemma." In *Genetic Algorithms and Simulated Annealing*, edited by L. Davis. London: Pitman.
- Belew, Richard K., John McInerney, and Nichol N. Schraudolph. (1992) "Evolving Networks: Using Genetic Algorithm with Connectionist Learning." In *Artificial Life II*, pp. 511-547, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Collins, Robert J., and David R. Jefferson. (1992) "AntFarm: Towards Simulated Evolution." In *Artificial Life II*, pp. 579-601, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Dawkins, Richard. (1989) "The Evolution of Evolvability." In *Artificial Life I*, pp. 201-220, edited by Christopher G. Langton. Redwood City, CA: Addison-Wesley.
- Dawkins, Richard. (1986) *The Blind Watchmaker*. New York: Norton.
- De Jong, K. A. (1975) "An Analysis of the Behavior of a Class of Genetic Adaptive Systems." Doctoral dissertation, University of Michigan, University Microfilms no. 76-9381.

References continued on next page.

References continued

- Fogel, D. B. (1997) "IEEE Transaction on Evolutionary Computation Editorial: Evolutionary Computation: A New Transactions." In *IEEE Transaction on Evolutionary Computation*, vol. 1, no. 1, p. 1.
- Fogel, D. B. (1994a) "An Introduction to Simulated Evolutionary Optimization." *IEEE Transactions on Neural Networks*, vol. 5, no. 1 (Jan. 1994), pp. 3-14.
- Fogel, D. B. (1994b) "Evolutionary programming: an introduction and some current directions." *Statistics and Computing*, vol. 4, pp. 113-129.
- Fogel, D. B. (1993) "On the Philosophical Differences between Evolutionary Algorithms and Genetic Algorithms." In *Proceedings of the Second Annual Conference on Evolutionary Programming*, pp. 23-29, edited by D.B. Fogel and W. Atmar. Palo Alto, CA: Morgan Kaufmann.
- Fogel, D. B. (1991) "The Evolution of Intelligent Decision Making in Gaming." *Cybernetics and Systems*, vol. 22, pp. 223-226.
- Freund, John E. (1962) *Mathematical Statistics*. Englewood Cliffs, NJ: Prentice Hall, Inc.
- Goldberg, David E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley.
- Hillis, W. Daniel. (1992) "Co-Evolving Parasites Improve Simulated Evolution as an Optimization Procedure." In *Artificial Life II*, pp. 313-324, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Jefferson, David, Robert Collins, Claus Cooper, Michael Dyer, Margot Flowers. Richard Korf, Charles Taylor, and Alan Wang. (1992) "Evolution as a Theme in Artificial Life: The Genesys/Tracker System." In *Artificial Life II*, pp. 549-578. edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Knuth, Donald E. (1973) *The Art of Computer Programming: Sorting and Searching*. Reading, MA: Addison-Wesley.

References continued on next page.

References continued

- Koza, John R. (1992) "Genetic Evolution and Co-Evolution of Computer Programs." In *Artificial Life II*, pp. 603-629, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Koza, John R. (1991) "Evolving a Computer Program to Generate Random Numbers using the Genetic Programming Paradigm." In the *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 37-44, edited by Belew and Booker. Palo Alto, CA: Morgan Kaufmann.
- Levinson, Gene. (1994) "Crossovers Generate Non-Random Recombinants under Darwinian Selection." In *Artificial Life IV*, pp. 90-101, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Levy, Stephen. (1992) *Artificial Life: The Quest for a New Creation*. NY: Pantheon.
- MacLennan, Bruce. (1992) "Synthetic Ethology: An Approach to the Study of Communication." In *Artificial Life II*, pp. 631-658, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Maley, Carlo C. (1994) "The Computational Completeness of Ray's Tierran Assembly Language." In *Artificial Life III*, pp. 503-514, edited by Christopher G. Langton. Reading, MA: Addison-Wesley.
- Morris, Harold C. (1989) "Typogenics: A Logic for Artificial Life." In *Artificial Life I*, pp. 369-395, edited by Christopher G. Langton. Redwood City, CA: Addison-Wesley.
- Nilsson, Nils J. (1980) *Principles of Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann
- Paredis, Jan. (1994) "Steps Towards Co-Evolutionary Classification Neural Networks." In *Artificial Life IV*, pp. 102-108, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Ray, Thomas S. (1998) "Selecting Naturally for Differentiation." In *Proceedings of Genetic Programming 1997 Conference*, pp. 414-421, edited by John R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, R. L. Riolo. CA: Morgan Kaufmann.

References continued on next page.

References continued

- Ray, Thomas S. (1994) "A Proposal to Create a Network-Wide Biodiversity Reserve for Digital Organisms." Paper distributed at the Fourth International Workshop on the Synthesis and Simulation of Living Systems. July 6-8, 1994 at MIT.
- Ray, Thomas S. (1992) "An Approach to the Synthesis of Life." In *Artificial Life II*, pp. 371-408, edited by Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. Reading, MA: Addison-Wesley.
- Reynolds, Craig W. (1994a) "An Evolved Vision-Based Model of Obstacle Avoidance Behavior." In *Artificial Life III*, pp. 327-346, edited by Christopher G. Langton. Reading, MA: Addison-Wesley.
- Reynolds, Craig W. (1994b) "Competition, Coevolution and the Game of Tag." In *Artificial Life IV*, pp. 59-69, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Sagan, Carl. (1995) "Where Did TV Come From?" *Parade Magazine*. p. 14. New York, NY: Parade Publications.
- Sipper, Moshe. (1994) "Studying Artificial Life Using a Simple, General Cellular Model." *Artificial Life*, pp. 1-35, vol. 2, no.1, Fall 1994.
- Stanley, E. Ann, Dan Ashlock, and Leigh Tesfatsion. (1994) "Iterated Prisoner's Dilemma with Choice and Refusal of Partners." In *Artificial Life III*. pp. 131-175. edited by Christopher G. Langton. Reading, MA: Addison-Wesley.
- Thearling, Kurt. (1994) Presentation at the Fourth International Workshop on the Synthesis and Simulation of Living Systems. July 6-8, 1994 at MIT.
- Thearling, Kurt, and Thomas S. Ray. (1994) "Evolving Multi-Cellular Artificial Life." In *Artificial Life IV*, pp. 283-288, edited by Rodney A. Brooks and Pattie Maes. Cambridge, MA: MIT Press.
- Tomita, Masaru. (1982) "Dynamic Construction of Finite Automata From Examples Using Hill-Climbing." In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pp. 105-108. Ann Arbor, MI: The Cognitive Science Society.

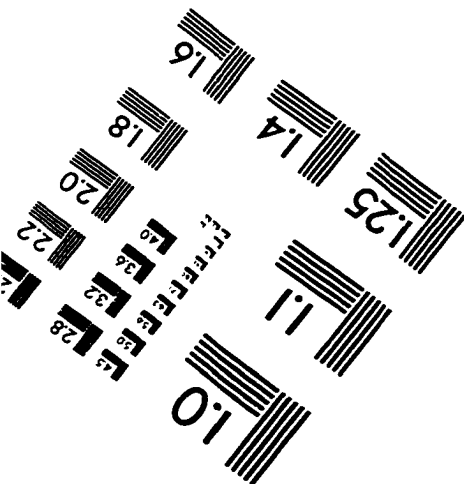
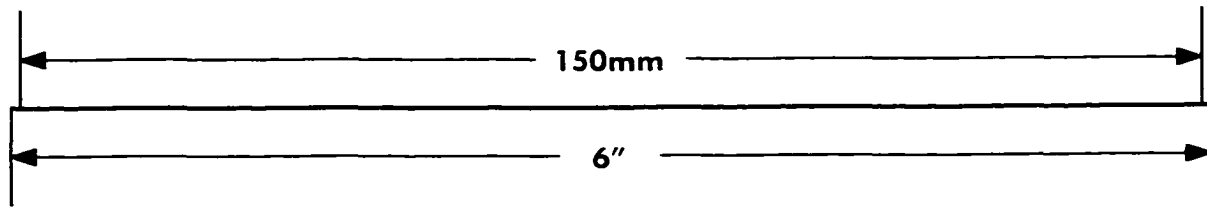
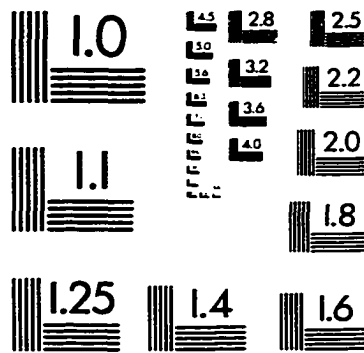
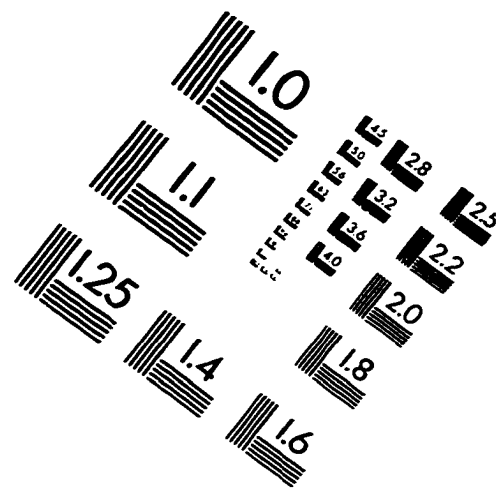
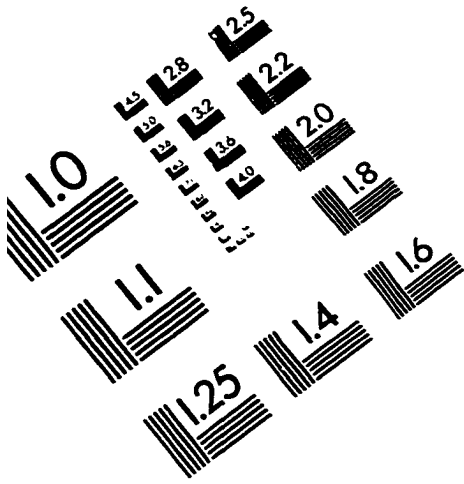
References continued on next page.

References continued

Whitley, Darrell (1989a) "The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best." In *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 116-121, edited by J.D. Schaffer. CA: Morgan Kaufmann.

Whitley, Darrell (1989b) "The Genitor Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. Technical Report CS-89-105." Fort Collins, Colorado: Colorado State University, Department of Computer Science.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

