Order Number 9315503

Joseki search, parallel computation and computer Go: A new approach to the Joseki problems

Shyu, Weng-Yu, Ph.D.

City University of New York, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

Joseki Search, Parallel Computation and Computer Go: A New Approach to the Joseki Problems

by

Weng-Yu Shyu

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.

1993

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

1/20/93
Date

Michael Anshel
Chair of Examining Committee

1/21/93
Date

Executive Officer

Dr. Stefan Burr

Dr. Jerry Waxman

Dr. Tien-Lih Teng

Supervisory Committee

The City University of New York

ABSTRACT

Joseki Search, Parallel Computation and Computer Go: A New Approach to the Joseki Problems

by

Weng-Yu Shyu

Adviser: Professor Michael Anshel

Now that startlingly successful computer chess programs have been developed [1] [2], a similarly advanced Go program is the next logical goal for the AI game-modeling community. A strong Go program was an original goal of the Japanese Fifth Generation project, but it was later dropped. To date, the best Go programs are still at the beginner level. The primary reason for this is that the game of Go has a much larger number of legal moves than does chess and it is also much harder to evaluate the strength of positions with heuristics. Furthermore, since most Go programs run on a personal computer, they lack necessary computational resources. Because the game tree generated by a Go game is so large, an exhaustive search is not possible with current technology. Adding more computational resources to current programs may enhance the abilities of these programs. Advances in the hardware circuits and the algorithms used to play Go - as we introduce here - make a far greater difference. In this

thesis, we develop a system to handle the Go opening game. Because of the multiplicity of choices, this requires a sophisticated searching method and much computational power. First, a pattern recognition program extracts the book knowledge called 'joseki' from a joseki dictionary. Second, a tree construction program compiles the knowledge into a tree. Third, an algorithm is used to match the pattern in a joseki with the pattern on the game board. Fourth, an alpha-beta search and a static evaluation select the best joseki for the next move. Some parallel evaluation algorithms under hypercube machines are also discussed. Finally, logic circuits are designed to implement some of the critical functions in the algorithm and their simulations are shown. These circuits dramatically improve the speed of the algorithm. They can also be applied to the middle game and the end game of a Go program.

ACKNOWLEDGMENTS

I deeply appreciate Professor Michael Anshel, my mentor, for his precious helps and guidance in these years. His inspiration and knowledge are invaluable resources. Everything I learned from him has become a great asset to me.

Most heartfelt thanks to Professor Tien-Lih Teng, Chairman of the Computer Science Department at the New York Institute of Technology, who kept the momentum behind me in these years. Without his helps and encouragement, this work would have been much harder to finish.

I would like to thank my committee, Professor Stefan Burr, Professor Jerry Waxman, and Professor Stanley Habib for their helpful comments and I also like to thank Professor Joseph Fulda who helped with the thesis preparation.

Finally, I thank my wife and parents for their support for my studies, always.

TABLE OF CONTENTS

Joseki Search, Parallel Computation and Computer Go:

A New Approach to the Joseki Problems

CHAPTER 1

Preliminary Concepts

The game of Go originated in China, subsequently spread to Japan and Korea, and is now played throughout the world. The basic rules of the game are very simple to automate [3] yet a powerful computational model of the Go game as a whole is extremely hard to develop. Several Go programs have been developed, but are considered weak. Most programs use the traditional game tree search and static evaluation methods to find the next move. In this chapter, the difficulties of constructing a powerful Go program and the problems in processing the opening game are discussed. We also review the development of programs for the Go game.

1.1 The Game of Go

1.1.1 Introduction to the Game of Go

Go is a perfect-information, two-person, zero-sum game. Two players, BLACK and WHITE, place their black stones and white stones at the intersections of a 19-by-19 grid. Players move alternately; a move consists of putting a stone onto the board. The main objects are to occupy as large a territory as possible and to capture as many of your opponent's stones as possible. Once a stone is placed, it is never moved unless it is killed and removed. The game is over when neither player can make a move to increase the size of his territory or capture any of his opponent's stones. A 19-by-19 array is used to simulate a real Go board. The matrix gives the origin at the lower left-hand corner of the board; the intersections of the game board at the lower left-hand corner is at (1,1) and the upper right-hand corner is at (19,19).

Figure 1.1 shows a Go game. Figure 1.1(a) shows an opening game, while Figure 1.1(b) shows a middle game. In Figure 1.1(c), the stones at the lower left-hand corner are in the end game, while the stones in the central area and the other corners are still in the middle game. Figure 1.1(d) illustrates an end game. From the figures, it is possible to

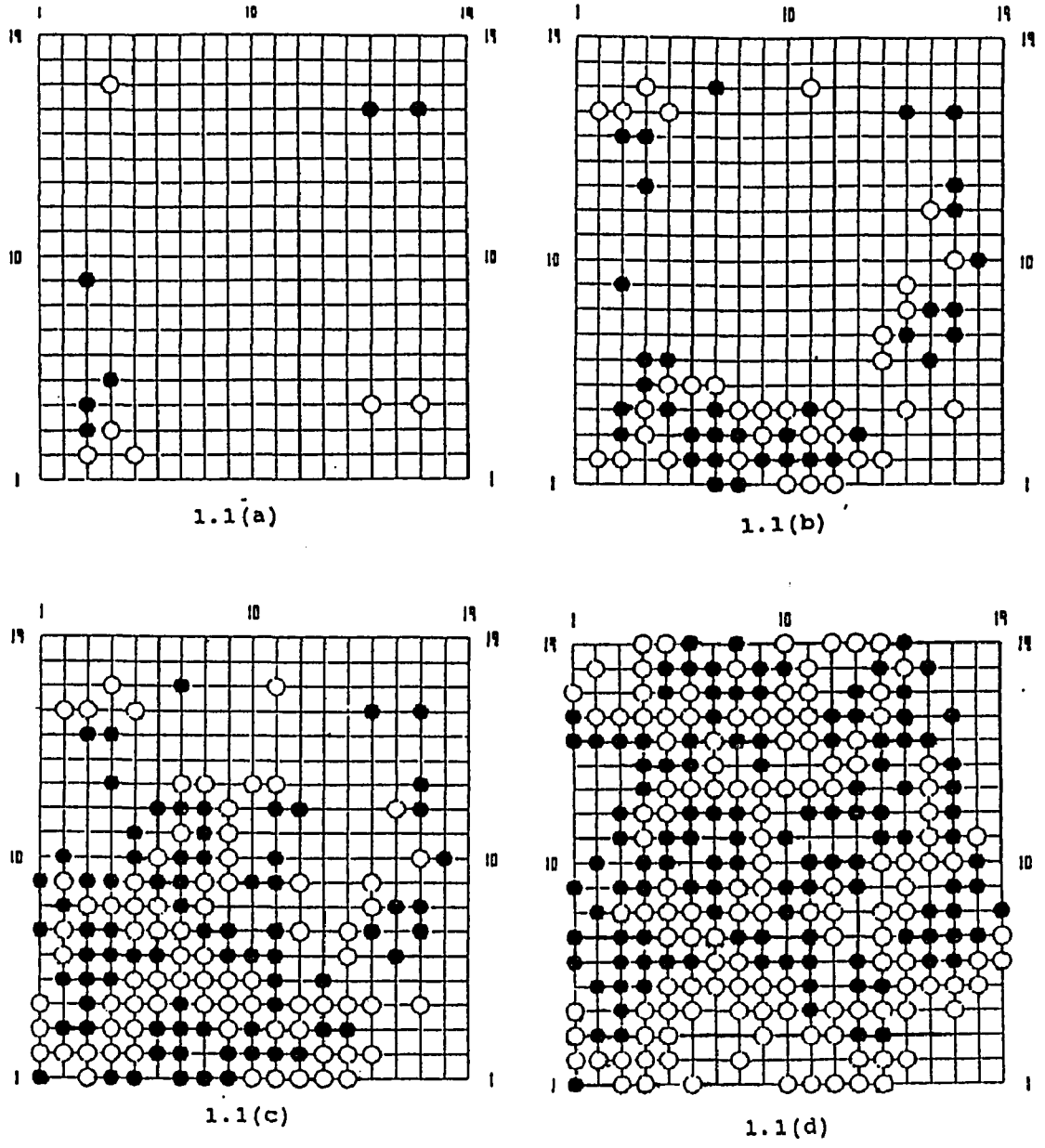


Figure 1.1 A game of Go.

see how battles may exist in different areas of the board requiring both players to switch their area of concentration from one location to another. All local battles are ultimately interrelated and may expand to affect other areas of the board. A group of stones in danger may expand in order to reach a group of stones which is safe.

Only stones next to each other horizontally or vertically are *adjacent*. Hence, stones on a diagonal are not adjacent. Adjacent stones of the same color connect together to form a single entity called a *link*. Any vacant points adjoining a link are called *liberties of the link*. A link of stones dies and is removed from the Go board when it lacks liberty; that is, when it is completely surrounded, within and without, by the enemy stones. It follows that the stones in a link all survive or die together.

Figure 1.2 shows some cases in which the white stones surround the black stones. There is no liberty for the black links in Figures 1.2(a)-(d). In these cases, the white stones capture the black links and remove them from the board. Although there are one or two remaining liberties for the black links in Figures 1.2(e)-(f), WHITE will eventually occupy them and the black links will die. In Figure 1.2(e), placing a white stone at (5,5) eliminates the liberty of the black link. Although the white stone at (5,5) lacks liberty

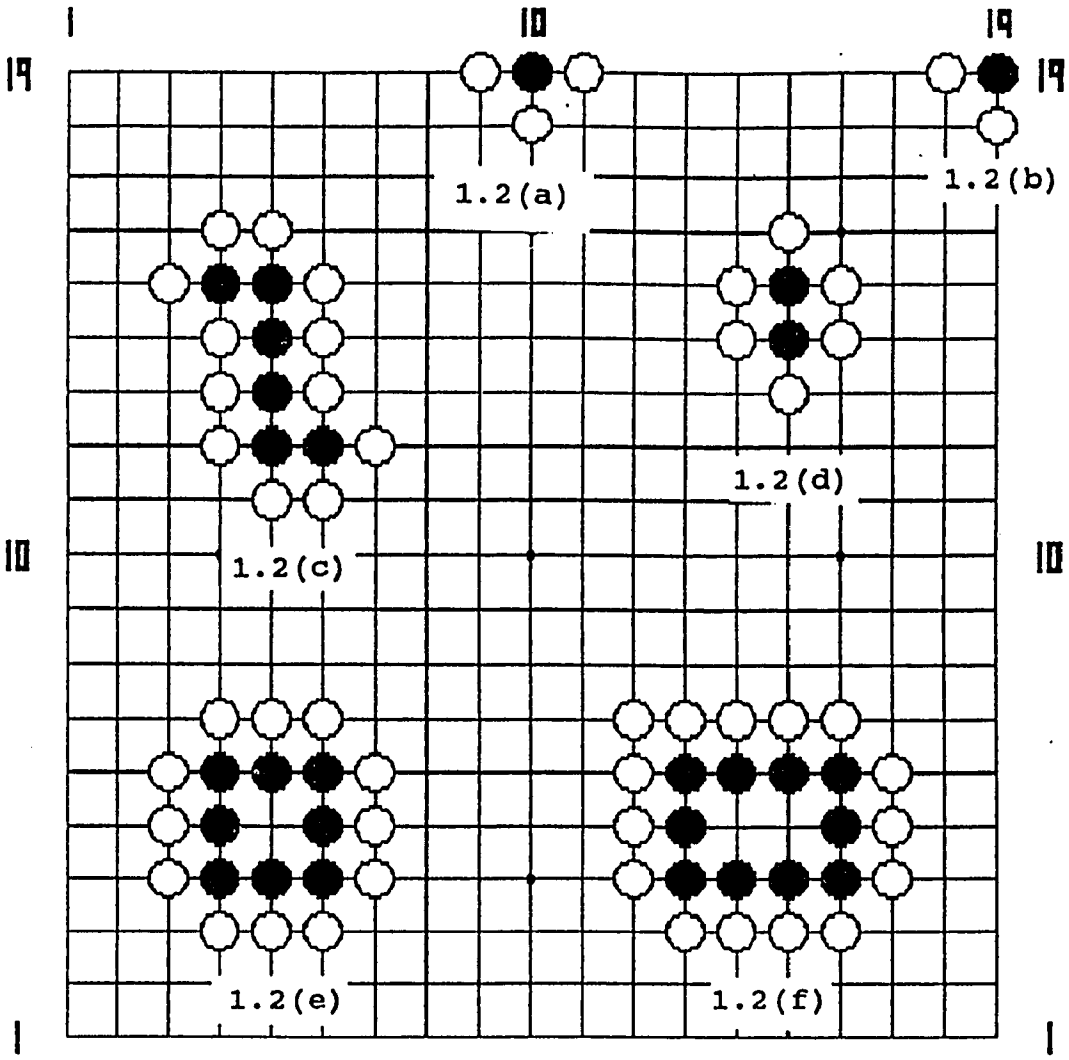


Figure 1.2 Dead black links in various configurations.

when it is placed at this intersection, this is not considered a suicide move. Instead, it causes the black link to be killed. Placing a stone in a particular location is forbidden only if it will have no liberties and will not be able to capture and remove an enemy link thereby creating liberties.

A *territory* is a vacant area which is surrounded by one player's stones and in which no enemy stones can survive. A territory can be subjected to challenge. In Figure 1.2(f), the black link surrounds two vacant points at (14,5) and (15,5) but they are not in BLACK's territory. WHITE could play at (14,5) and (15,5) to kill the black link. If BLACK plays at (14,5) after WHITE plays at (15,5), WHITE can immediately place a stone at (15,5) again, killing the black link.

A link of stones is guaranteed survival only if they have at least two secure, separated liberties. The black links illustrated in Figure 1.3(a) and Figure 1.3(b) have such liberties. In Figure 1.3(a), the link connects to two secure liberties called *eyes*. No matter which eye WHITE places his stone in, the black link still has a liberty. Placing a white stone in such an area without somehow creating liberties is considered a suicide move, and is forbidden. In Figure 1.3(c), two eyes shared by two black links and the two links

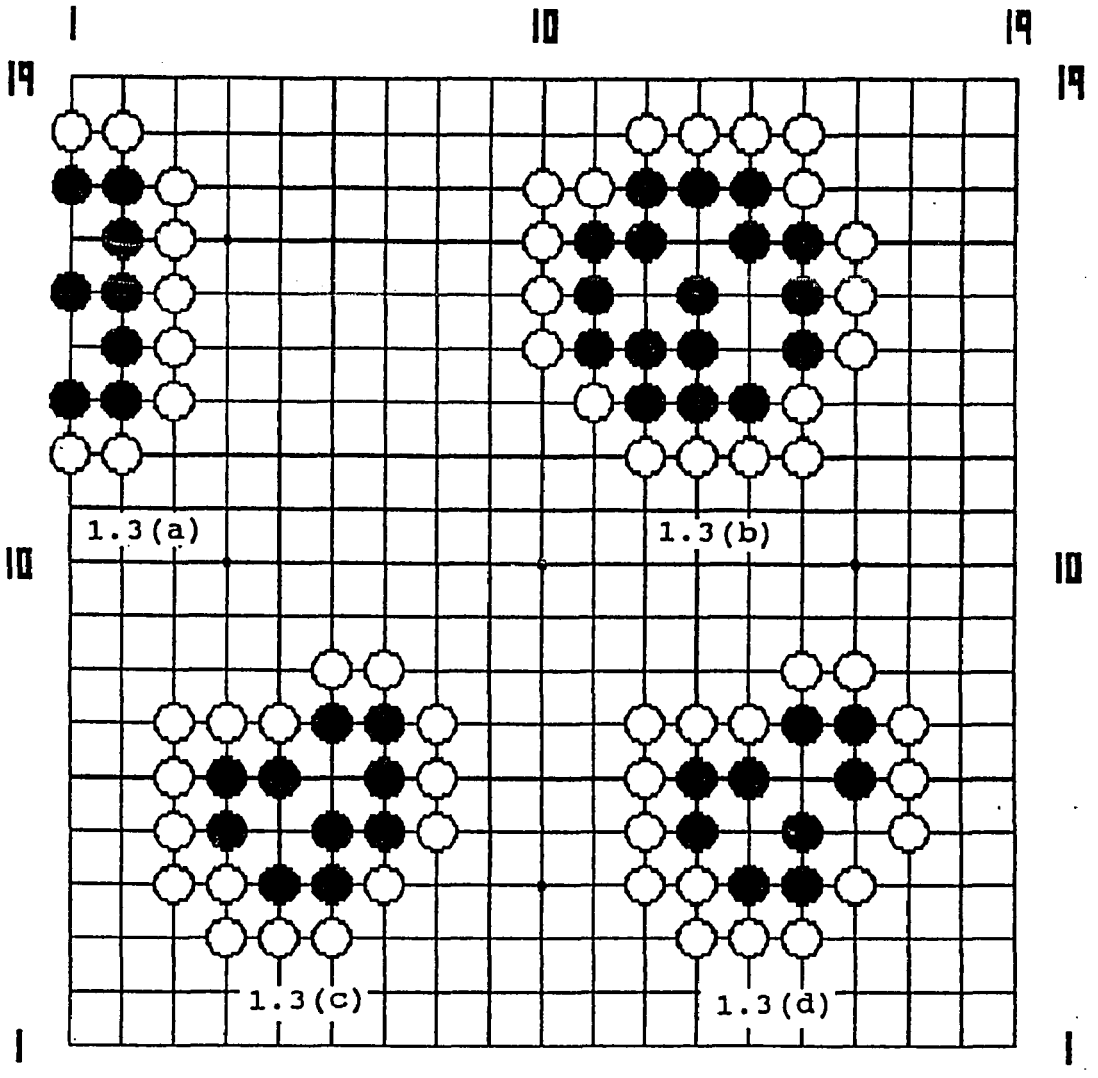


Figure 1.3 Eyes and false eyes of black links.

themselves constitutes the eyes together. Both links are secure.

In Figure 1.3(d), the black stones have two eyes. If WHITE is next to play, these two eyes will be considered *false eyes*, that is, they may be destroyed by attacking the link surrounding them. WHITE must first play a stone at (16,5) isolating the link at the locations (15,7), (16,7), and (16,6). Then a stone played at (15,6) captures it. Finally, an additional white stone at (14,5) kills all the black links.

The patterns shown in Figures 1.3(c) and 1.3(d) are similar to each other except for one position. That position generates a totally different outcome. A player has to inspect the stones carefully to decide on a move. Meanwhile, on the other side of the board, there are several local battles occurring. As they spread, the outcome of these battles may affect the other groups on the board. A player must co-ordinate his efforts in different areas in order to take control of large territories. Hence, he has to base his decision on a move not only on the local situation but also on its impact on the entire game board.

At the end game, each captured stone counts as 1 point. An empty intersection in the controlled territory also counts as 1 point. And, the player who has the most points wins the

game.

Normally, a game of Go starts from any of the four corners of the game board. The corners are the easiest locations for a group of stones to create enough vacant points to ensure survival. The corners are also the easiest territories to occupy: It takes three stones to securely occupy the vacant point at the corner (1,1) or (19,19) but five stones to occupy a vacant point on the edge of the game board and eight stones are needed to enclose a vacant point at all other locations. After building a group of stones at a corner which have enough eyes to ensure survival even under attack, the player tries to extend the influence of this group of stones into border areas, which are the next easiest areas in which to create more territories. In addition to gaining advantage from the edge, the stones near the edges also cooperate with other groups of stones in trying to enclose the central area. Hence, the central area is the last area to be considered.

The players normally begin by placing the stones on the lines which are three or four lines away from the edges, i.e. x=3, x=4; x=16, x=17; y=3, y=4; y=16, y=17. Placing a stone at the line x=3 allows a player to control the area between x=3 and the edge x=1, though it extends less power in the central area. Placing a stone at the line x=4 introduces more power in the central area, but control of the area between x=4

and the edge x=1 is less secure and more easily intruded upon. At all times, groups of stones at different locations have to cooperate.

Rankings for beginning and intermediate amateur Go players begin at 35 kyu, the rank assigned to novice players, and go to 1 kyu. Rankings then continue from 1 dan upward to a maximum of 9 dan. At least two years of serious play and study are required to reach 1 dan.

1.1.2 The Difficulties Inherent in Designing a Powerful Go Program

A brute-force search chess program generates a game tree for look-ahead evaluation. It explores all the possible ways of playing a game and evaluates these paths. The brute-force approach searches all the paths in the game tree except the nodes known by the algorithm to be valueless. The brute-force approach plays excellent tactical chess because it can see all outcomes that can take place within several moves. On the other hand, it fails to understand the importance of the long-range strategic aspects of the game.

To get a strategic result, a program must either evaluate the game thoroughly statically or dynamically search the tree in depth. Since the size of the game tree in a brute-force search grows exponentially, it cannot explore the game tree

deeply enough to affect the strategic result. Since there are so many nodes to be evaluated, only a simple evaluation for each node can be given.

A program which evaluates a game thoroughly has bigger problems than that of the brute-force search approach. The evaluations normally take large amounts of time and the results of evaluations sometimes conflict with each other. Quick evaluation of a game and coordination of the evaluation results are complicated when many evaluations are involved.

Using a brute-force search and a little knowledge has proven to be very effective in computer chess [1]. A chess-playing machine, a combination of software and customized hardware called Deep Thought, defeated a chess grandmaster in 1988. Some differences between Go and chess shed light on why it is so difficult to computerize Go successfully:

1. Too many branches in each level of the game tree

In Go, there are 361 legal positions at the beginning of the game. Normally, it takes approximately 200 moves to complete a game. Even at the end of a game, the number of legal moves remaining is too high for a program to handle all possibilities. It has on average about 6 times more legal moves than chess.

2. Deep search is required

It is common for a Go player to examine a line of play going 15-20 moves deep. Such in-depth analysis is rare for a chess master. Often, it is impossible for a program to explore a tactical result by exhaustive tree search because the game tree is not deep enough. The strategic advantage of a Go game is achieved only slowly by successively adding stones to a large board. A shallow search can hardly find a good move.

3. Go game is difficult to evaluate by a program

When a chess piece is lost in play, it normally represents a significant disadvantage. Some properties of chess allow the board to be evaluated according to clear criteria such as a point system based on the pieces present. On the other hand, there is no such pre-dominant factor controlling the Go game. Life and death of a group is an important criterion, but the death of a group of stones often does not mean a loss. It is common to sacrifice a stone to gain a positional advantage that may not be seen immediately. The tactical results of Go do not override the strategic outcomes. Even more, the precise life and death evaluation is a hard problem which has yet to be solved.

Because of the size of the Go board, the game is fragmented into several small battles. Players must spread their attention around the various battles. Evaluations must be performed on each of these battles separately and then the interactions between them must be determined in order to find the best overall position. Furthermore, the variety of strategies needed to achieve a target makes the evaluation even harder. To occupy more territory than one's opponent, one may attempt to build a wall to enclose more territory, attack a weak enemy link, or intrude on the opponent's territory. These strategies change during the course of the game. Evaluating the pattern on the game board by some simple evaluation function does not do. The importance of a static evaluation function also depends on which stage of the game is being simulated.

In a Go game, the positional advantage and strategic move both play an important role. Looking ahead 10 to 20 moves during a game is common to decide the consequences of a move. A game tree which uses brute-force to search for the next move has around 150^{10} nodes to be evaluated; too many and yet a little knowledge is not sufficient to decide a positional or strategic advantage statically. Computer Go is thus harder to computerize than chess and is still in its infancy.

1.2 Joseki

1.2.1 Introduction to Joseki

A *joseki* is a fixed sequence of moves which are considered locally equitable at the corner for both players. Thousands of josekis have been discovered and accumulated since the Go game was invented. Figure 1.4 is a typical joseki figure in a Go joseki dictionary. The unnumbered stones represent the initial condition called the *base figure*. Under this condition, the stones 1 through 4 show the orders and the positions one should play to lead to the joseki. The patterns that are generated by a base figure and the stones from 1 to m are called *joseki patterns*. m is the greatest number less than or equal to the maximum number associated with the stones of the joseki figure. In Figure 1.4, the base figure and the stone 1 generate a joseki pattern. The base figure and stones 1 and 2 also generate a joseki pattern. The base figure in a joseki normally is generated by a sequence of moves from another figure. A figure that has a non-empty initial condition is a continuation of another figure. The sequence of moves which generate a joseki is called a *joseki sequence*.

All josekis could be collected and represented by a tree. In our program, each node in the tree containing the

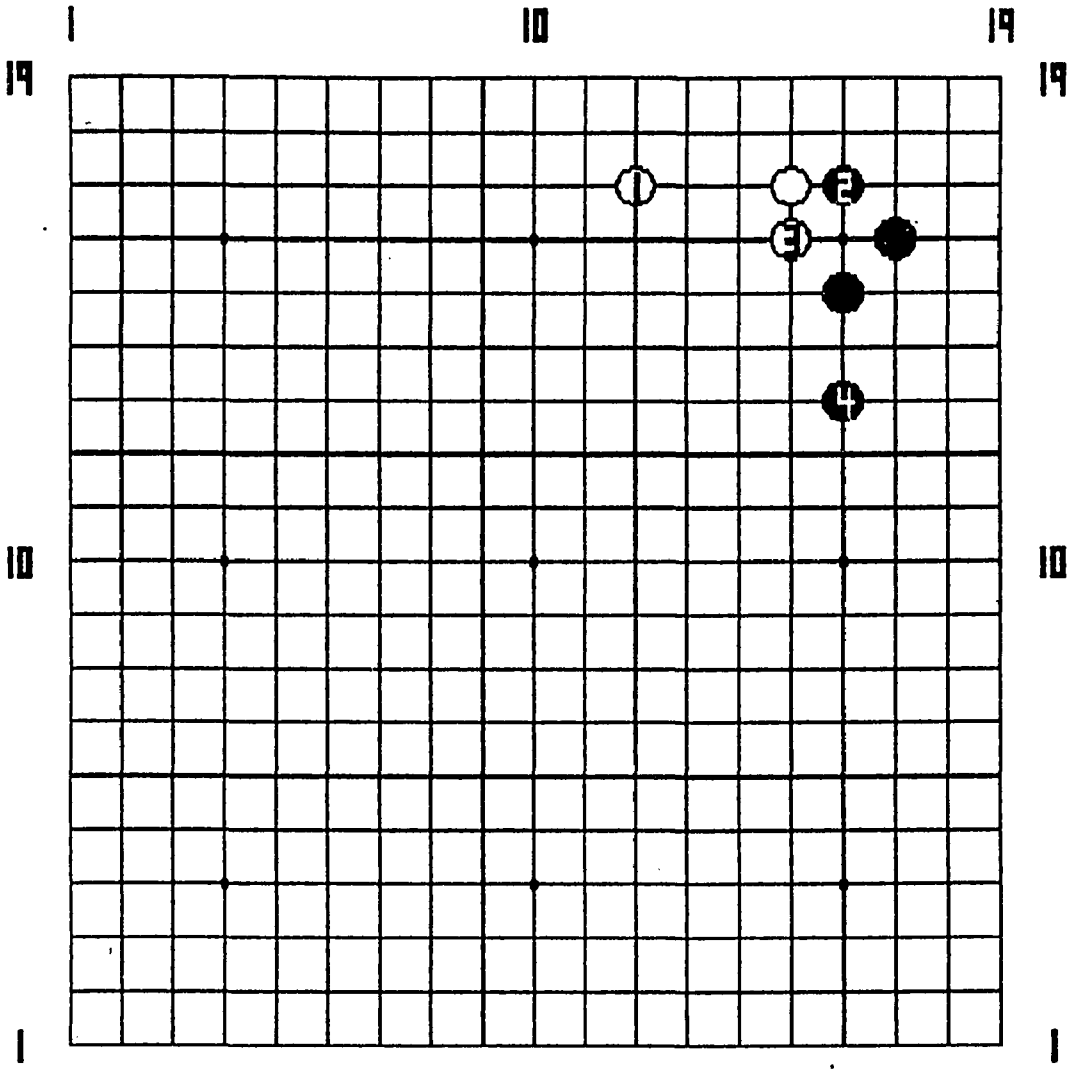


Figure 1.4 A joseki figure.

(x,y) coordinates, a color, and some properties represents a move in a joseki sequence. A node is said to *represent* a stone in the joseki sequence. Placing the stones represented by the nodes in a path from the root to any node in the tree generates a joseki pattern. One node in the tree may have several branches representing the different joseki sequences derived from one joseki pattern. If a node contains the (x,y) coordinates, color c, c being either BLACK or WHITE, and a stone with color c is at the location (x,y), then we say that the node *matches* the stone.

1.2.2 Issues in Joseki Processing

To use josekis properly, one must recognize a joseki and select a move appropriate to the overall situation. Wilcox [4] raised the following issues in handling a joseki move:

1. If a joseki appears after a sequence of non-joseki moves, how does a program recognize that a joseki situation has just appeared?

2. How much of the board is to be considered? A joseki may extend from one side of the board to the other, as shown in Figure 1.5.

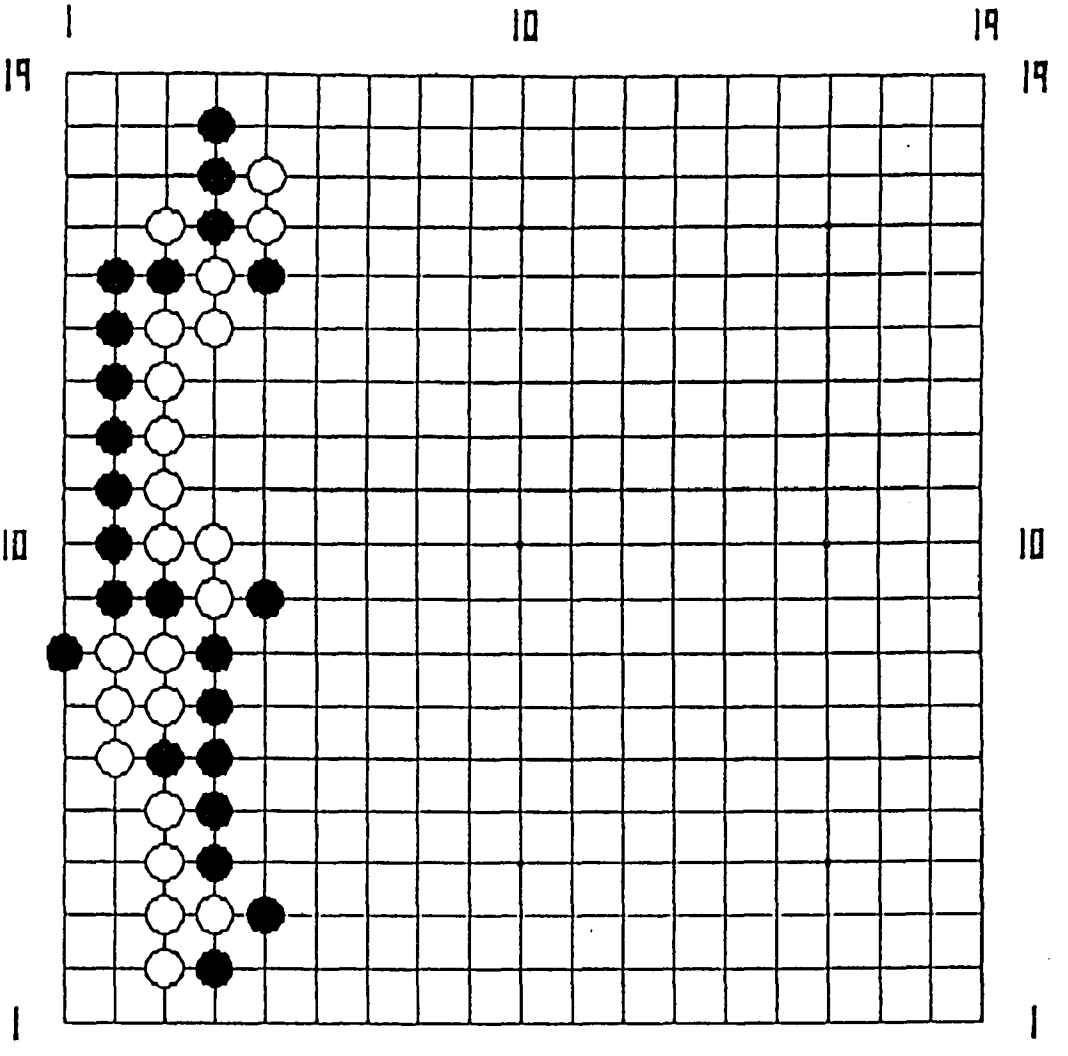


Figure 1.5 A joseki spans the game board.

3. How does a program recognize a joseki move in any of the four corners, each of which has two orientations and two colors to deal with?

4. How does one handle a move after a joseki?

5. How does one handle as many as four josekis in play concurrently - one joseki for each corner? (The joseki database is shared.)

6. When do other stones invalidate joseki knowledge?

7. How does one select a move appropriate to the overall situation?

If both players follow a joseki sequence, the joseki tree and a pointer which points to a node in the tree can be used to trace the moves. The pointer always points to the node that represents the most recent joseki move. The joseki move can then be found easily by obtaining the coordinates from one of the child nodes. Chen [5] used a joseki tree and four pointers that point to the current move of the joseki being played for four quadrants. A joseki pattern may originate from a non-joseki sequence. If a sequence of moves, say, 1, 2, 3, 4 is a joseki sequence and the moves 3, 4, 1, 2 are not a joseki sequence, these two sequences generate the same

joseki pattern. A pointer tracing down the joseki tree cannot recognize a joseki pattern generated from a non-joseki sequence because the non-joseki sequence does not exist in the joseki tree. A joseki pattern should therefore be recognized from a raw Go pattern.

A joseki is a local situation. To recognize a joseki from a raw Go pattern, only part of the entire game board should be considered. Given a set of stones on the game board, the problem lies in how to isolate an area from the entire game board so as to recognize a joseki. Some josekis stay only in a 9-by-9 area. Other josekis may extend to the other side of the game board, as shown in Figure 1.5.

A joseki dictionary shows joseki sequences in isolation. No other stones are near the area where the joseki exists. In a real game, there may be stones around a joseki. Some of them may be fairly close to the joseki. These stones become *interferences* to the joseki. A joseki pattern under interferences is shown in Figure 1.6. All the stones excepting the one at (2,16) belong to a joseki. The stone at (2,16) may become a fatal interference to the joseki. Sometimes, a stone stays close to a joseki but it is not a fatal interference. Therefore, the interferences of a joseki must be examined carefully.

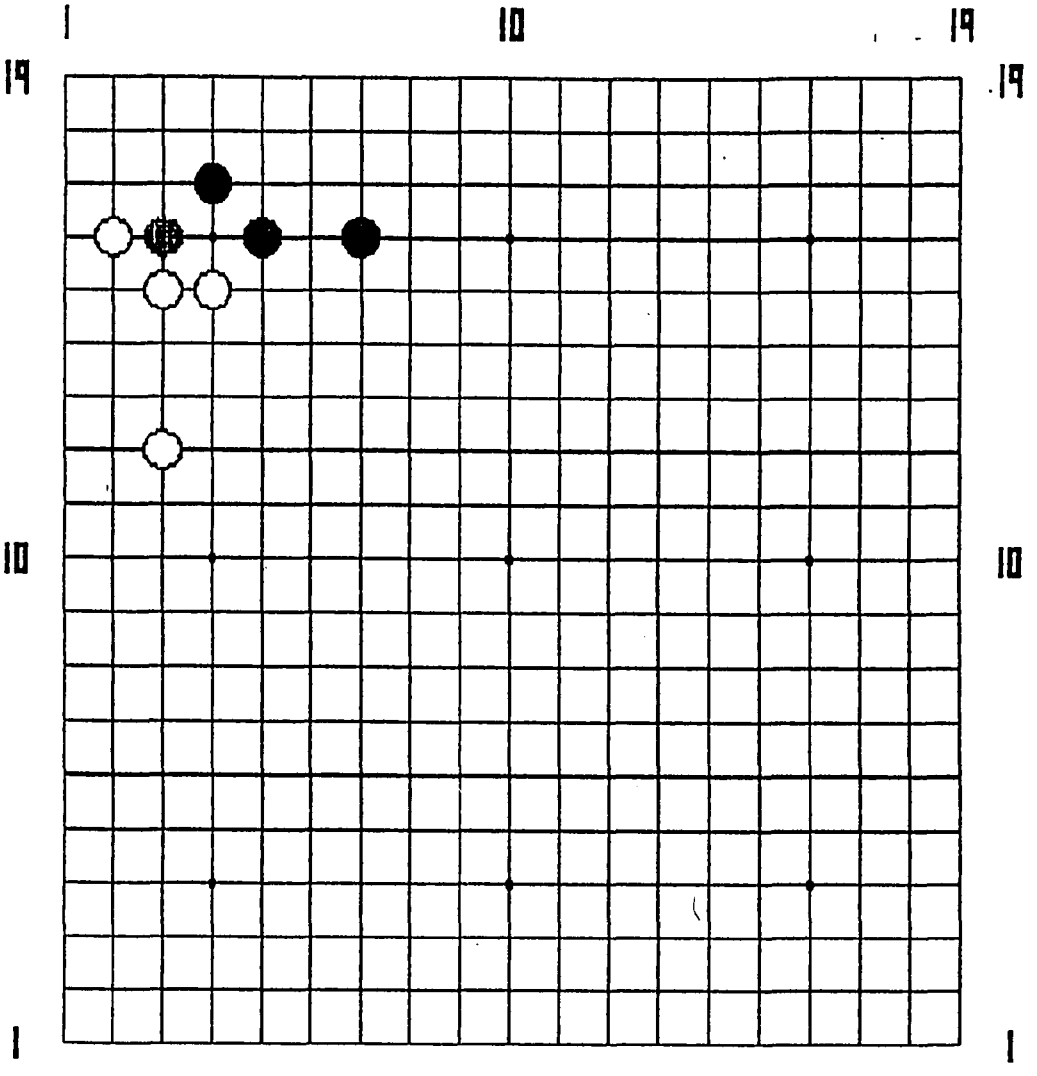


Figure 1.6 The grayed stone at (2,16) interferes with the joseki.

It also should be noted that several joseki sequences may derive from a joseki pattern, but not all of them are equally good for a game. A joseki only takes care of local situations. Selecting a joseki optimal to a global situation is another important problem.

Using josekis requires considerable knowledge. Today's programs just don't have enough ability to use josekis properly [6]. Figure 1.7 shows the opening game, Goliath *vs.* Go Intellect, played in the 1990 US Computer Go Championship [7]. Goliath was the first-place program and Go Intellect the second-place program in the tournament. Both programs are rated close to single kyus. Go Intellect is poor in corner playing. There is still much to be improved in joseki automation.

1.3 The Development of Go Programs

1.3.1 Zobrist's Go Program

Zobrist's program was the first created for Computer Go [8]. Pattern recognition is the main mechanism used to find the next move. A set of patterns called *configurations* is predefined and matched with every position, orientation, and reflection of the board. When a match is found, a score is

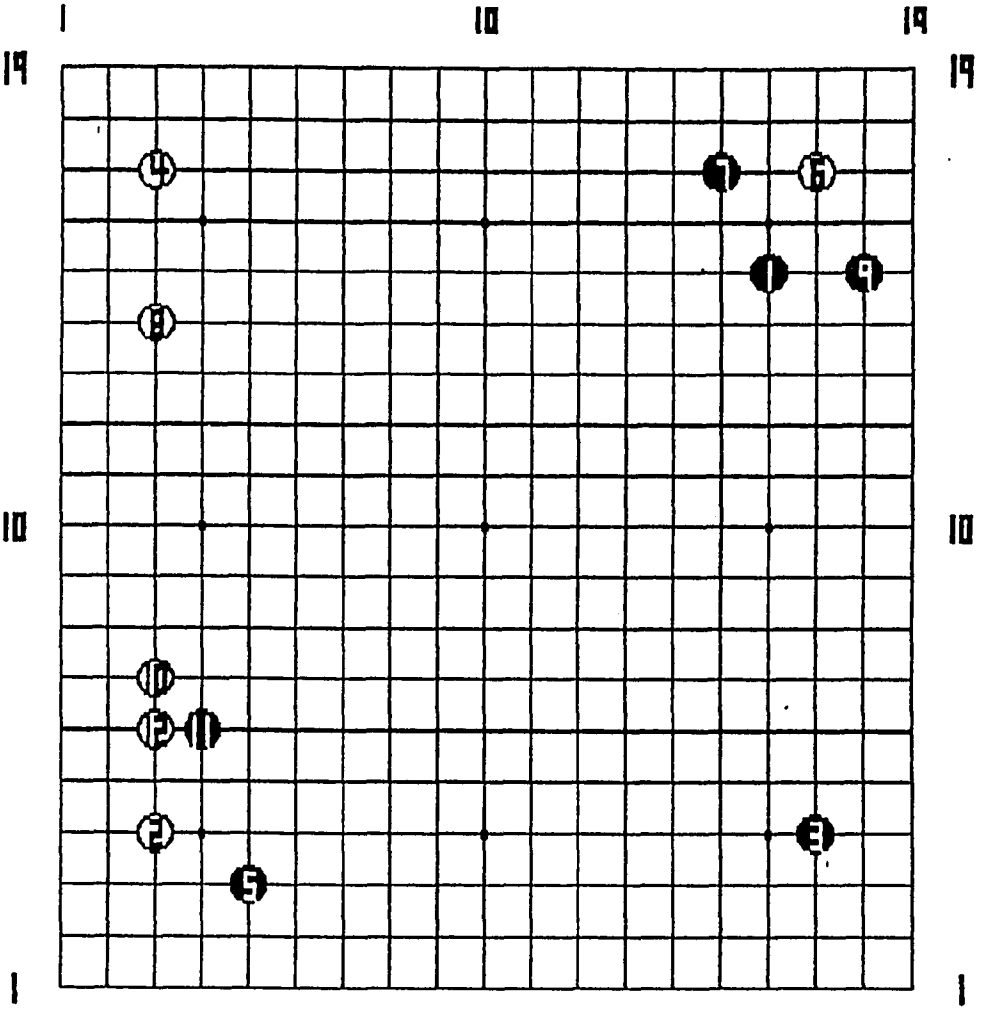


Figure 1.7 The opening game - Goliath vs. Go Intellect.

assigned to some position. For example, a configuration contains the following information:

(0,0) has a black stone,
(1,0) is a vacant intersection,
(1,1) is a safe black stone,
(1,0) is weighted 500.

The relative location (0,0) is the current position under examination. Several scores may be assigned to a position after all patterns are examined.

Look-ahead Mechanisms

A look-ahead or tree-search mechanism is used for the following goals:

(1) capture a chain or save a chain from capture,
(2) connect two chains,
(3) prevent two chains from connecting,
(4) threaten an army of stones with capture.

There are 27 configurations that are used to select candidates for the look-ahead procedure. Instead of assigning a weight to a position, these configurations specify numbers 1 or 2 in some locations. A game-tree search is then performed in these locations. At the first level, the look-

ahead mechanism will examine the possibility of the move at the intersections that correspond to the numeral "1" marked by the configurations. For the other levels, look-ahead is restricted primarily to the area corresponding to the numerals "1" or "2". The look-ahead calculations are confined to a 5-by-5 area centered at the average of the two moves at the two previous levels. The look-ahead mechanism only examines to a depth of three plies and is extremely weak. If a move is found for a goal, e.g., save-a-chain-from-capture, a weight is added to the location.

The scores from the pattern match and look-ahead procedures are summed up for each position. The position with the highest score is the program's next move.

Zobrist's program was estimated to be at around 31 or 32 kyu, the equivalent of a human who has played 15 to 20 games.

1.3.2 Ryder's Go Program

For most Go programs, the "influence function" is an important building block. It is used to define a fighting unit, handle potential territory, and identify a strong or weak control area. An area is strongly controlled by one side if it is hard for the other side to survive in the area. Usually, this area is easily occupied by the side with

stronger control. Figure 1.8 shows an example of the influence function: The locations nearest the stone have the greatest number; other locations have numbers which decrease exponentially with distance. Black stones radiate positive numbers, while white stones radiate negative numbers. The influence of a position is the summation of all influences on the position radiated from all stones. The influence value in a position shows the degree of control of one side. Figure 1.9 shows the accumulated influence generated by the stones on the board.

In Ryder's Go program [9], the entire game board is organized into stones, strings, groups, and armies. A string consists of a single isolated stone or two or more stones of the same color located on immediately adjacent grid points. A wall is an area strongly controlled by one player. An army consists of the stones in an area which have influence greater than a user-defined value. A wall and an army are both defined by influence. Figure 1.10 shows how the program generates a move. First, the program determines the current status of all strings on the board. A developmental analysis is then performed at every legal move followed by a tactical analysis and rudimentary strategic analysis, on which immediate elaboration.

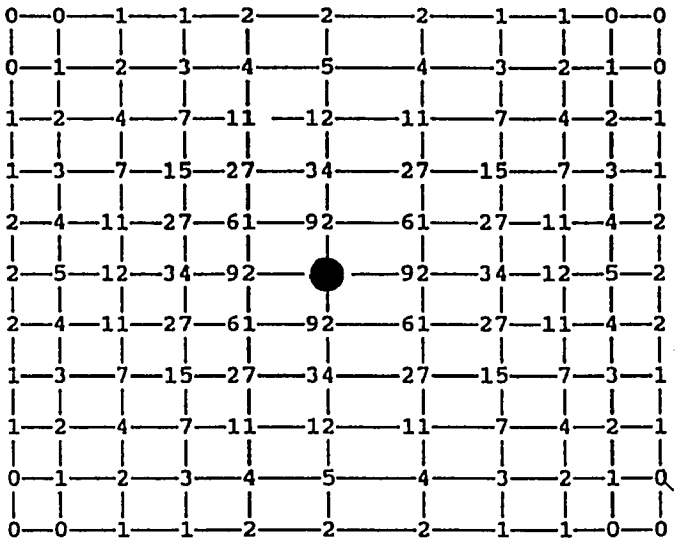


Figure 1.8 An influence function of a black stone.

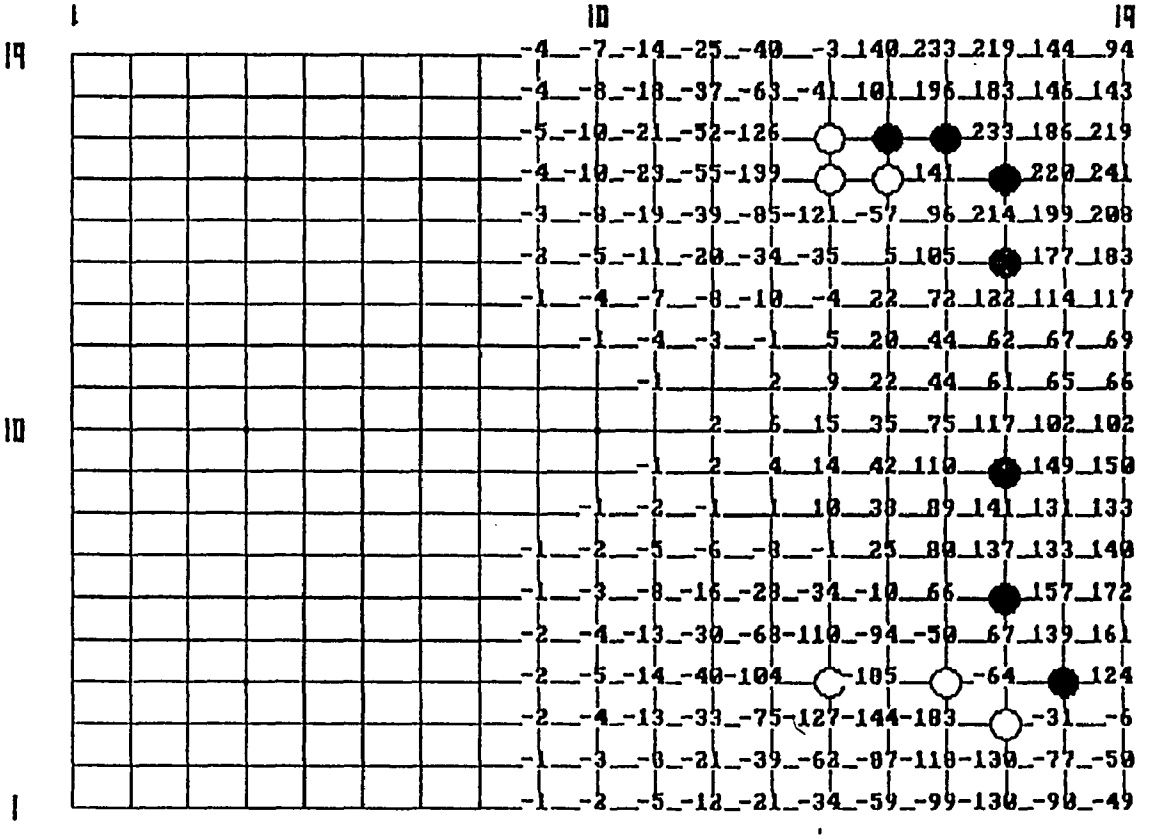


Figure 1.9 The accumulated influences generated by the stones on a game board.

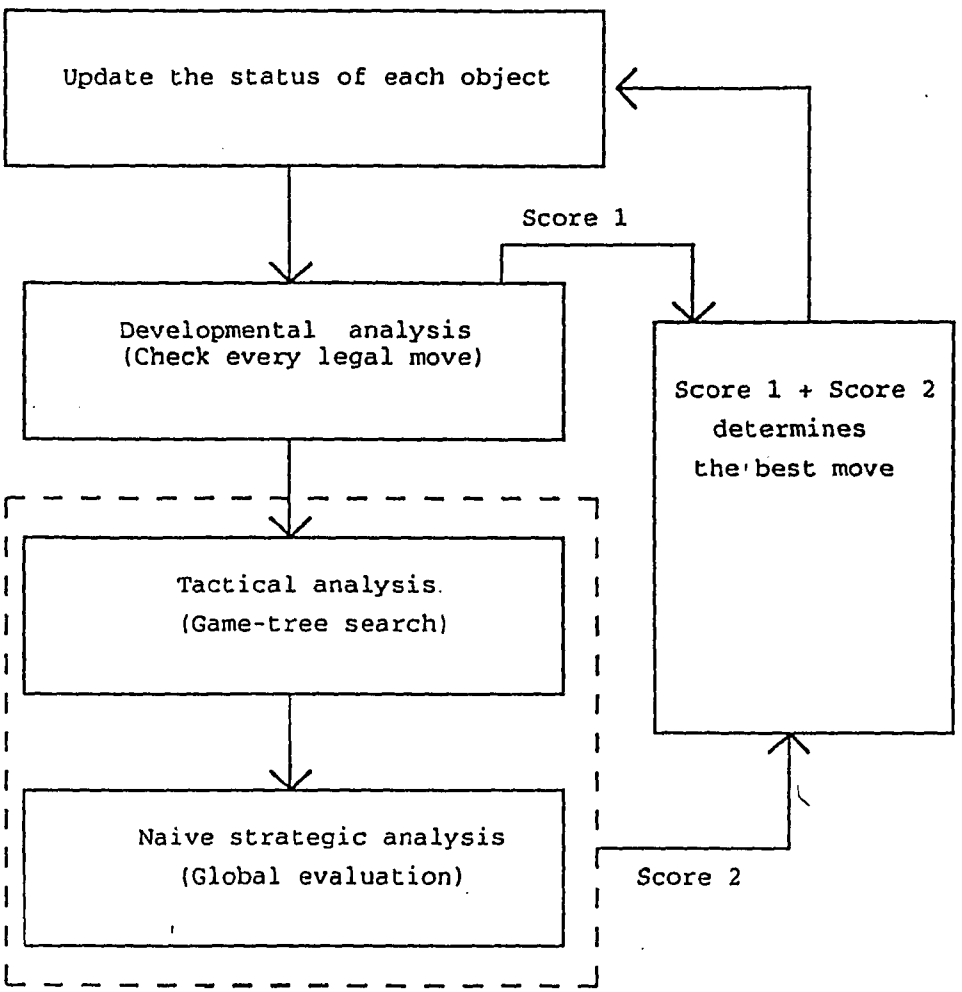


Figure 1.10 The move decision process of Ryder's program.

Developmental Analysis

In a Go game, a move can serve several purposes. Among these, it may strengthen a friendly group, kill enemy stones, or threaten to enclose enemy stones. A good move may accomplish two or more such goals simultaneously. Developmental analysis measures a set of properties that can be developed or enhanced by a move. These properties represent an attempt to accomplish some goals. Examples of these parameters are:

1. a move makes more area for the enemy,[1]
2. an enemy move combines enemy groups,[2]
3. a friendly move reduces some enemy stones to only two liberties.[3]

There are 35 such properties to be checked in the analysis. After the developmental analysis, each legal move is given a preliminary evaluation. Normally, a move's score is higher if it possesses more properties in the analysis. The fifteen highest- scored intersections are selected for the second-pass analysis.

---

[1] It creates more territories for the enemy.

[2] The enemy group is bigger and harder to catch.

[3] They are now an easier target to capture.

Tactical Analysis

In the second pass, the tactical analysis and rudimentary strategic analysis are applied. For each move generated from the developmental analysis, the tactical analysis procedures analyze the potential capture of strings which have small numbers of liberties after the move is put on the board for evaluation. A minimax tree search is applied in the analysis. Only "live" and "death" values are generated at the leaf evaluation. The search generates move candidates dynamically for each level. Among all move candidates, normally only the 4 best nodes are selected for further expansion. If the tree grows beyond 10 levels, only the two best nodes are selected.

When the tree generation stops, the last node generated is considered a leaf if cutoff conditions occur. Cutoff conditions include:

1. there are too many strings involved,
2. the tree is expanded beyond fifty nodes,
3. the string under inspection has more than five liberties.

Rudimentary Strategic Analysis

In the strategic analysis, the status of armies, walls, groups, and territories are measured. The resulting evaluation is an estimate of the overall positional security of either side. There are 34 parameters in the analysis. These include:

1. number of armies,
2. number of walls,
3. influence of all groups,
4. number of small groups,
5. number of endangered groups.

Group safety is evaluated in the strategic analysis. This is an important parameter in game playing and is defined by the following terms:

1. number of eyes,
2. number of group points in the group,[4]

---

[4] Ryder describes group points as follows:

"A group is a set of contiguous points which are all connected or half-connected to the same side....

A point is connected to BLACK (say) if either of the following conditions is true:
1. the point is occupied by a live black stone, or
2. a. the point is not occupied, and
   b. the point is directly adjacent to at least one black stone, and
   c. the point is not adjacent to any white stones.

A point is half-connected to BLACK (say) if all the following conditions are

3. number of area points in the group,

4. whether or not the group is surrounded,

5. number of moves connecting to other groups,

6. number of moves extending the group.

The values generated by the tactical analysis and the rudimentary strategic analysis are added to the scores from the developmental analysis. The move with the highest total score is the move recommended by the program.

1.3.3 Review of INTERIM and INTERIM.2

Walter Reitman and Bruce Wilcox found that the pattern-directed inference process plays an important role in Go. Their program consists of a multilevel pattern recognition system. The basic elements on a board are organized into strings, chains, territories and groups. A chain is formed when two strings of the same color, in close proximity, with no intervening enemy stones, are linked. A group is one or more strings of the same color that attacks and defends as one. [10][11][12]

---

true:
1. the point is not occupied, and
2. the point is directly adjacent to at least two black stones, and
3. the point is directly adjacent to at least one white stone."

The main pattern recognition components

The pattern recognition system consists of lens, edge and web components. The *lens* system recognizes and monitors local stone configurations. It recognizes the position and the orientation of the particular pattern in the board and associates it with the information stored in the program. It monitors the linkages, corners and patterns near the edges of the board. The scope of the lens is no more than 25 board intersections. The lens system also analyzes the similarity of a pattern to the standard patterns whose properties are known. The scope of the lens may overlap.

The *edge* component takes care of the corners and edges. It measures such parameters as the number of vacant points along the edge between a stone and its neighbors to detect the empty regions on the corners and edges. The program will then use this information to propose an extension of a friendly group or an invasion of an enemy group.

The *web* component provides all the proximate, tactical data about a group. This information is collected by inspecting the positions around the group, layer-by-layer, starting from the positions neighboring the group.

Move selection in INTERIM.2 program

Procedure 1.1 shows how INTERIM.2 generates its moves. The INTERIM.2 program first updates the information about links, groups, local patterns, surrounding situations, and sector lines. PROBE, a look-ahead program, generates information in a tactical data structure. LOCAL.URGENT and its subfunctions deal with urgent problems that may be raised by the consequences of an immediately preceding move pair. Only a handful of thousands of josekis stored in a joseki lens are accessed by PLAY.JOSEKI. If there is no move for LOCAL.URGENT, the program will try to find a move to satisfy the goals in DEVELOP.GROUP, ATTACK.GROUP, RUN.TOWARD.DEAD.GROUP and other routines (see Procedure 1.1).

INTERIM 2.3 was ranked at approximately 15 kyu. At least four to nine months of serious play and study are required to reach 15 kyu.

1.3.4 Review of "Go Explorer"

The relationships between stones are categorized into 3 types of frames: blocks, chains, and groups. Each has eight to thirty properties. The local urgent pattern, patterns which require an immediate response to avert massive losses,

```
MOVE
     UPDATE.POINT.TYPES
     UPDATE.STRINGS
     UPDATE.LENSES
     UPDATE.LINKS
     UPDATE.GROUPS
     UPDATE.WEBS
     UPDATE.SECTOR.LINES
     UPDATE.TERRITORIES
     UPDATE.TACTICAL.ANALYSES
               [PROBE]
     UPDATE.GROUP.STABILITY.ESTIMATES
REFLEX
     LOCAL.URGENT
               PLAY.JOSEKI
               STRING.ATTACK.AND.DEFEND
               LINK.ATTACK.AND.DEFEND
               RE.EDGE.LINK
                     [PROBE]
               VITAL.SHAPE.POINT
               CONTACT.FIGHT
     DEVELOP.GROUP
               KILL.CUTTING.STRING
               EXTEND.AND.SQUEEZE
               COUNTERATTACK
               CROSS.SECTOR.LINE
               STABILIZE.POTENTIAL.TERRITORY
               RUN.TO
     ATTACK.GROUP
               WIPE.OUT
               SHAPE.POINTS
                    [PROBE]
               SQUEEZE
               ENCLOSE
     RUN.TOWARD.DEAD.GROUP
     EXTEND.SECURE.INVADE
     DEVELOP.POTENTIAL.TERRITORY
     ENDGAME
```

Procedure 1.1 The move decision process of INTERIM.2.

are identified by the pattern recognition procedures. Josekis are stored separately as a tree, while the knowledge needed to generate moves to achieve specific goals are implemented by implicit production rules. Go Explorer evaluates a game situation using the present board, not a move history. It is rated around 15 kyu. [6]

1.4 Recent Developments

Here we outline recent software approaches to Go that are relevant to our investigation.

1.4.1 Review of "Go Intellect"

Go Intellect [13] is a second generation of Go Explorer [6]. It won first place at the Second Computer Olympiad in London and second place at the 1990 US Computer Go Championship. Figure 1.11 shows the move decision process of Go Intellect. The stones are organized into blocks, chains, and groups. A block is a "link" - a directly connected set of stones of one color. A chain is a collection of inseparable blocks of one color. A group is a collection of stones connected via chains and/or spaces with influence above a specified threshold. Each block, chain, and group has about 30, 10, and 50 slots, respectively, filled by some evaluation

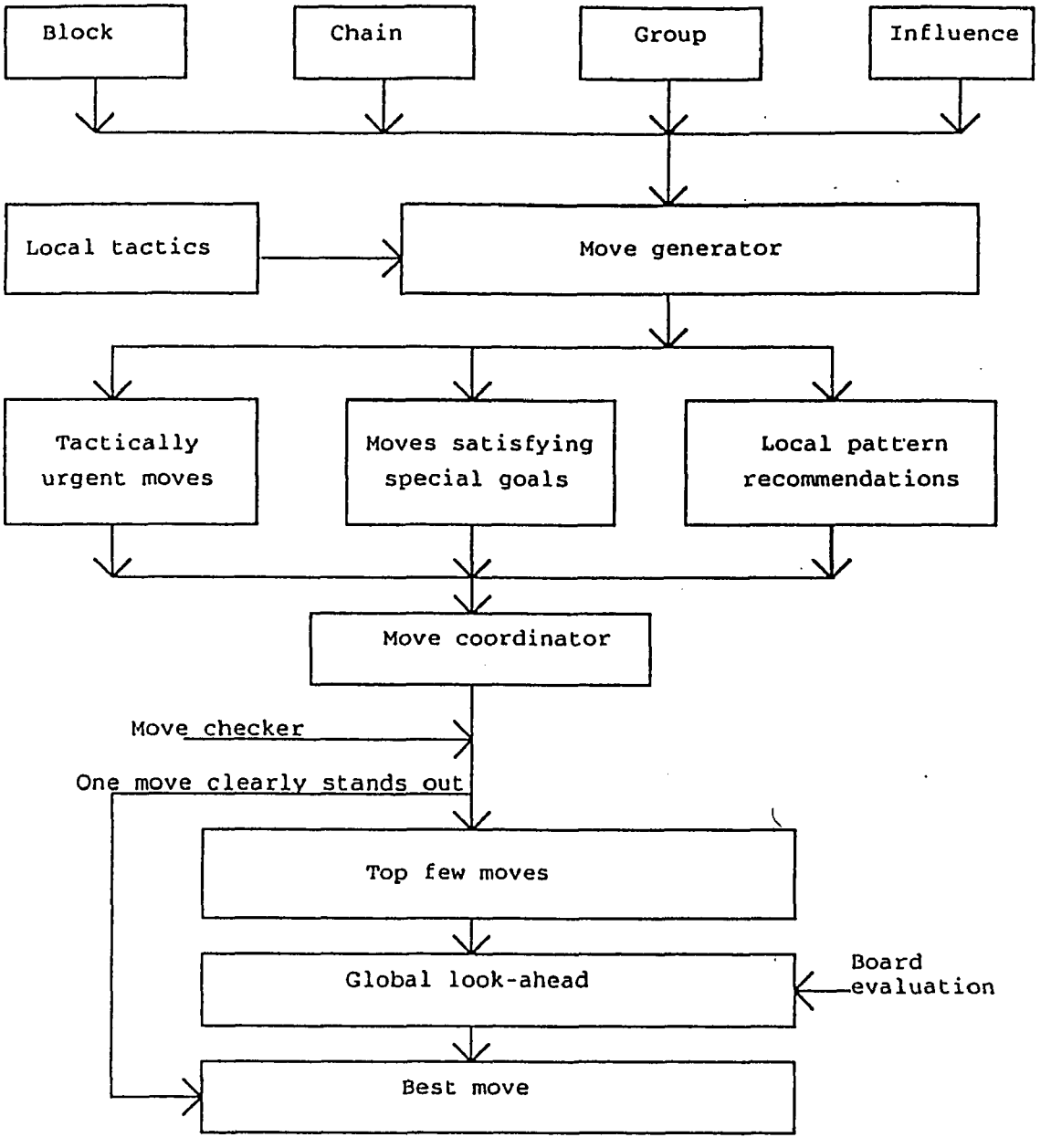


Figure 1.11 The move decision process of "Go Intellect"

values. Go Intellect has about 20 move generators, most goal-oriented and heuristic. Each move generator generates zero or more moves, each associated with a value. Two pattern libraries and a pattern matcher also generate a large set of moves with pattern values. The move coordinator combines all the values of a possible move using certain linear combinations. The move checker looks for obvious pitfalls of a candidate move.

The program determines the next move by comparing the value of different possible moves. If there is one move with a significantly higher value than any other move, this is the next move of the program. Otherwise, a few top-ranked moves are chosen for global look-ahead evaluation. The candidate moves of each level in the look-ahead procedure are generated from the raw board condition. Go Intellect does not consider more than 6 or 7 choices at each level, and does not go beyond depth 6.

1.4.2 A Joseki Processor

Bruce Wilcox [4] created a joseki processor in a commercially available program, NEMESIS™, for IBM PC compatibles. The processor recognizes a joseki from the board image. It then computes the maximum size rectangle needed to cover all moves in a joseki sequence to decide what stones

interfere with the joseki. As a pedagogical tool, Wilcox's program provides three ways to explore josekis:

(1) When a user picks up one of the available moves and plays it, the program displays the choices for the next player.

(2) The program can place a completed joseki after another deriving from the current position.

(3) It selects randomly from the choices but does not show the joseki to the user. If the user selects a move, the program accepts the move and generates the next one.

There is, however, no global evaluation for selecting a suitable move based on the overall condition. Moreover, in the paper, no detailed algorithm is revealed for recognizing a joseki.

In another study, Chen [5] also proposed a tree structure to store the information in a joseki dictionary. Each node in the tree consists of the current-move positions, next-move pointer, alternative-move pointer, color, and current-move properties. A pointer points to the current move of the joseki being played in each of the four quadrants. A minimax procedure can then be used to search for the best joseki for the current board configuration.

1.4.3 The End Game

A breakthrough in the Go end game, the only one advanced thus far, was announced by Berlekamp in 1990 [14]. His solution uses a combinatorial algorithm instead of the traditional AI game-tree search and evaluation.

CHAPTER 2

The Joseki Tree

2.1 Creating a Joseki Tree

The information about the josekis is collected and represented in a tree called a *joseki tree*. A node in the tree represents a stone in the corresponding joseki sequence. The joseki tree is implemented as an array, each entry in the array representing a node in the tree. Each node uses indices to point to its first son, its next brother, and its father node. A joseki tree is defined with the following C code:

```
struct nodetype {
                    int xy;
                    int father, son, brother;
                    int properties; } josekitree[1500];
```

The integer "xy" stores the x, y coordinates of a stone in the joseki sequence in 2 bytes with the first byte for x and the second byte for y. "Father", "son", and "brother" store the indices pointing to the father node, son node, and brother node. A node points to its first son with its son index. The other sons are linked by the brother indices of the nodes'

left brothers. The "properties" field stores the properties of a node including the color of the node. Figure 2.1 shows the structure of the tree.

Figure 2.2(a) and 2.2(b) show two josekis, A and B. Figure 2.2(a) has a joseki sequence (17, 16, black), (15, 17, white), (16, 15, black), (13, 16, white), (17, 10, black), and (9, 17, white). Figure 2.2(b) also has a joseki sequence (17, 16, black), (15, 17, white), (16, 15, black), (12, 17, white), (16, 17, black), (15, 16, white), (16, 13, black). Figure 2.3(a) shows the tree that represents these two josekis. The first 3 stones in the sequences are the same for both sequences; the remaining stones belong to the Fig. 2.2(a) and Fig. 2.2(b) sequences, respectively. The array representation of the tree is given in Figure 2.3(b). An entry represents a node in the tree. An entry consists of several fields: x, y, father, son, brother, and property. To construct a joseki pattern, a program follows the nodes in a path (as shown in Fig. 2.3(a)) and places the stones represented by the nodes onto the game board. The last node placed on the game board *defines* the joseki pattern.

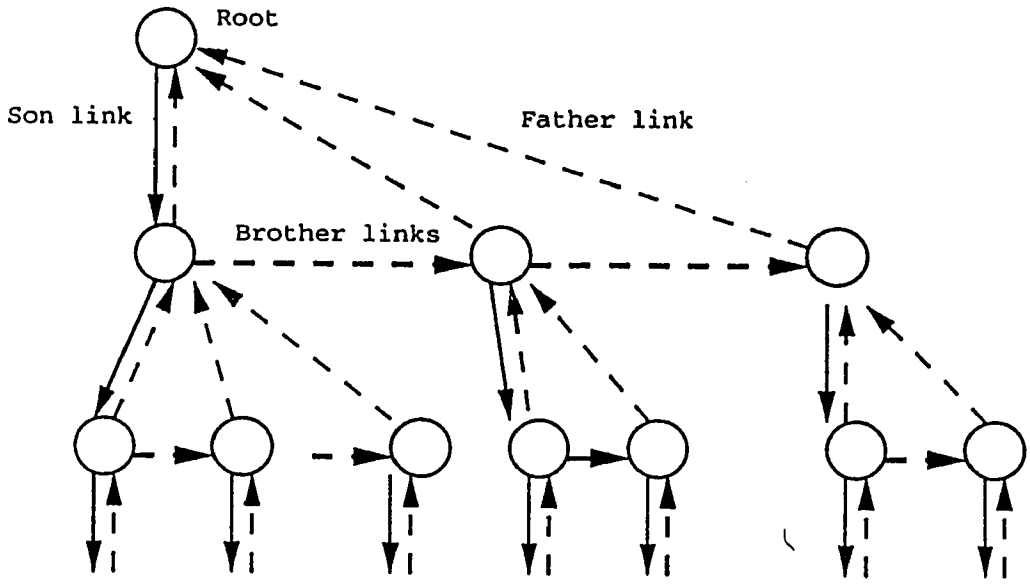


Figure 2.1 The structure of the joseki tree.

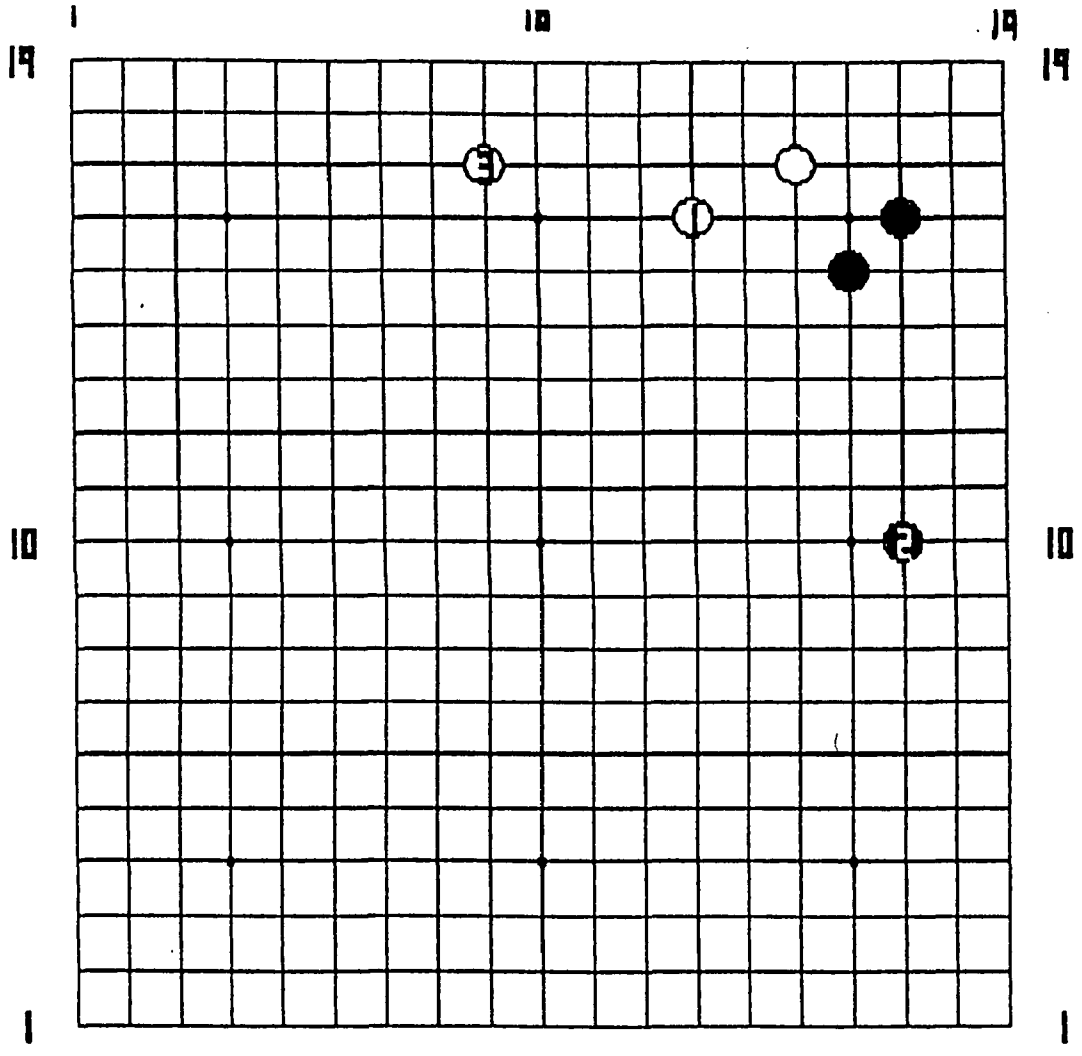


Figure 2.2(a) Joseki A.

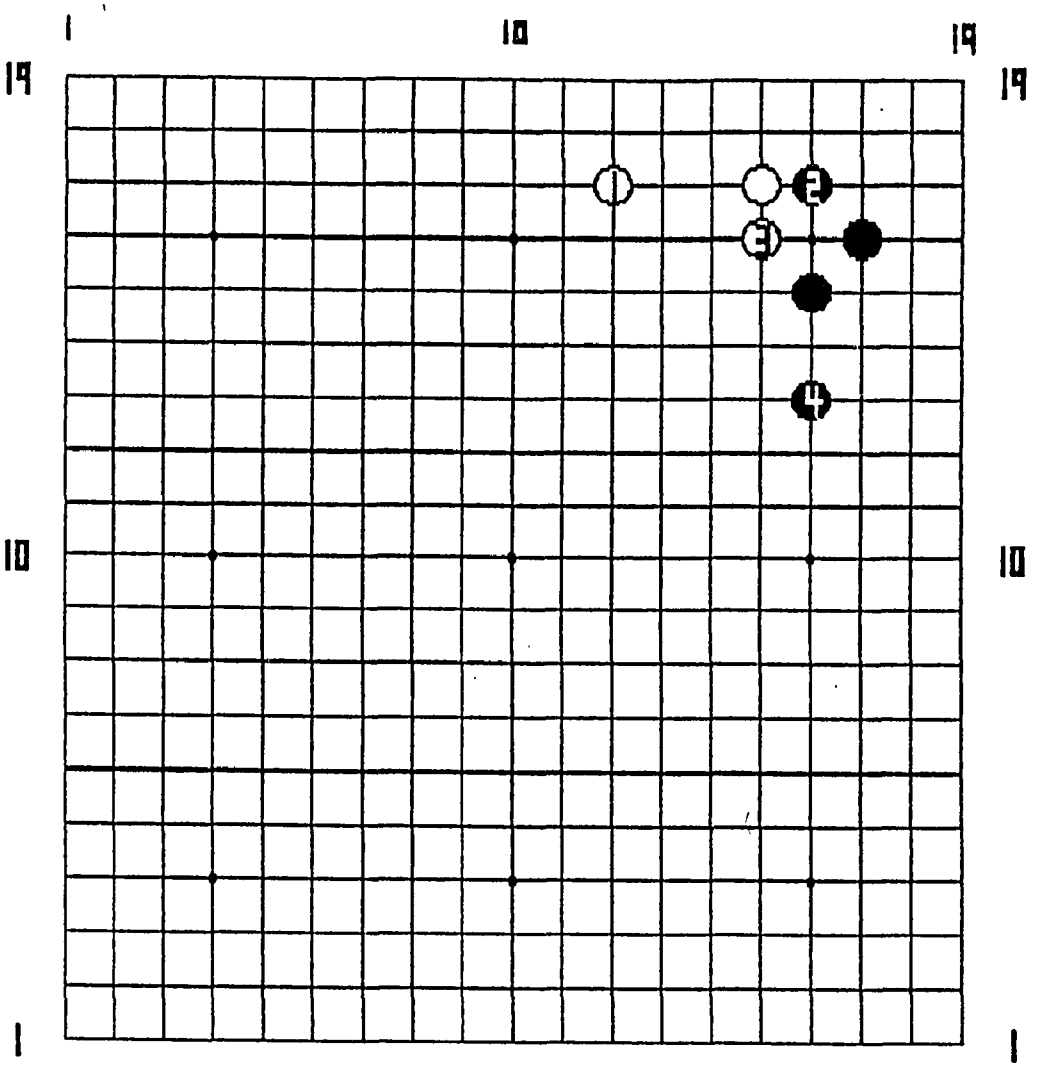


Figure 2.2(b) Joseki B.

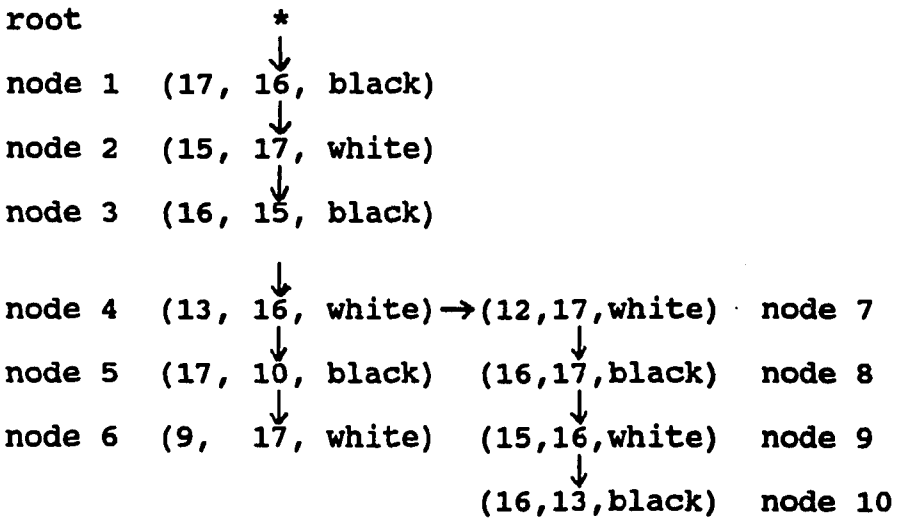


Figure 2.3(a) Tree derived from josekis A and B.

0	0,0,-1,-1,-1,-1
1	17,16,0,2,-1, 0
2	15,17,1,3,-1, 1
3	16,15,2,4,-1, 0
4	13,16,3,5, 7, 1
5	17,10,4,6,-1, 0
6	9,17,5,-1,-1,1
7	12,17,3,8,-1, 1
8	16,17,7,9,-1, 0
9	15,16,8,10,-1,1
10	16,13,9,-1,-1,0

Figure 2.3(b) Array implementation of the tree.

2.2 Extract the Josekis from the Joseki Dictionary

2.2.1 A Pattern Generator

Generating a joseki tree starts from determining the positions, color and number of the stones from the *joseki figures* in the joseki dictionary [15]. A program called PATTERN_GENERATOR was created to recognize the patterns in a joseki figure. Figure 2.4(a) shows a joseki figure. It has the exact size that we used in PATTERN_GENERATOR to extract the joseki information. It generated the information in a form suitable for the next program TREE_BUILDER which actually constructs the joseki tree. A joseki figure may have "empty" patterns, letter patterns ('a', 'b', 'c' ...), or patterns for numbered and unnumbered black and white stones.

Each joseki figure in the joseki dictionary is scanned by a black-and-white scanner. The scanner generates a bit pattern. Part of the bit pattern of a joseki figure is shown in Figure 2.4(b). The figure was generated by a scanner with 400 pixels per inch vertically and horizontally. 0 represents a black pixel and 1 represents a white pixel. PATTERN_GENERATOR applies the algorithm given in Procedure 2.1 to extract the positions and the numbers on the stones from the bit pattern automatically.

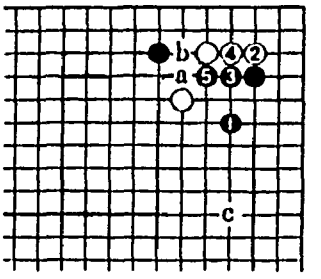


Figure 2.4(a) A joseki figure in the joseki dictionary.

```
00000000001111111111111111111100111111111111001111111111111111
00000000001111111111111111111100111111111111001111111111111111
00000000001111111111111111111100000000000000000000111111111111
00000000001111111111111111111100000000000000000000000000000000
00000000001111111111111111111100111111111111001111111111000000
00000000001111111111111111111100111111111111001111111111111111
00000000100111111111111111111100111111111111001111111111111111
00000001100111111111111111111100111111111111001111111111111111
00000001100011111111111111111100111111111111001111111111111111
00000011110011111111111111111000111111111111001111111111111111
00000111111001111111111111111000111111111111001111111111111111
00001111111100011111111111100011111111111111001111111111111111
00011111111110001111111000001111111111111111001111111111111111
01111111111111000000000000111111111111111111001111111111111111
11111111111111110000000011111111111111111111001111111111111111
00111111111111111000000011111111111111111000000011111111111111
00011111111111000000000000111111111111000000000000011111111111
00001111111110000000000000011111111100001111111100011111111111
00000111111100000000000000001111111100011111111111000111111111
00000011111000000000000000001111111000111111111100011111111111
00000001110000000000000000000111100011111110000011110001111111
00000001100000000000000000000011100111111100000111100111111111
00000000100000000000000000000011000111111000000111111001111111
00000000000000000000000000000010011111100000001111110011111111
00000000000000000000000000000010011111100000001111110011111111
00000000000000000000000000000010011111000000001111110001111111
00000000000000000000000000000010011111000000001111110001111111
00000000000000000000000000000010011110000000000111110011111111
00000000000000000000000000000010011000000000000111110011111111
00000000000000000000000000000010011000000000000111110011111111
00000001100000000000000000000011000100000000000111100111111111
00000001100000000000000000000011100111000000000011100111111111
00000011110000000000000000000011100011111110000111110011111111
00000111111000000000000000000011111000111111100011110011111111
00001111111000000000000000000011111100011111100111100011111111
```

Figure 2.4(b) Bit pattern generated from a scanner.

Locate the center positions of all intersections on the line x=7 that is the leftmost column on a joseki figure.

For every intersection on the line from the line x=7 to the line x=19,

Find the approximate center position of the intersections;

Find the number of black pixels on the 4 corners of the intersections;

Decide if there is any circle pattern around the intersection;

If there are few or no black pixels and no circle patterns exist,

Find the precise center position for the intersection,

Mark the intersection EMPTY;

If there are many black pixels and no circle pattern exists, a letter is in the intersection.

Mark the intersection ABC.

Otherwise, a stone pattern is on the location.

For each intersection that has a stone or letter on it,

Using the positions of those neighboring intersections interpolate the center position for the intersection with the stone or letter on it;

Specify the color of the stone if there is one at the intersection;

Compare the pattern around the intersection with the standard patterns and determine which number is on the stone or which letter is in the intersection.

Procedure 2.1 The algorithm for recognition of a stone pattern.

The algorithm first separates the entire input pattern into 156 subpatterns. Every subpattern has one of these: a white stone, a black stone, a letter or simply empty. The subpatterns' center positions are approximately at the intersections. For the algorithm to succeed, the center of every intersection must be found as precisely as possible. The algorithm first finds the approximate positions of intersections by using the center locations of the subpatterns. A stone pattern has a circle around it. Because circle patterns are less sensitive to the center position of an intersection, PATTERN_GENERATOR tries to identify a circle pattern first. It then separates those with circle patterns and those without circle patterns - i.e. empty intersections and letter patterns.

After the empty intersections are identified, the precise center position of the empty intersections are located. They all have a cross. The intersection of the cross is the center position. The center positions of the stone patterns and letter patterns are located by interpolation from nearby intersections. Empty intersections often serve as the precise bases of these interpolations. After the center location is found, bit-map matching is used to identify the letters, stones, and numbers.

2.2.2 Pattern Recognition for Josekis

The center of an empty intersection in the input pattern is not immediately obvious as shown in Figure 2.4(b). The sensitivity of the scanner (which can be adjusted) and the quality of the printing in the joseki dictionary affect the quality of the input patterns. The darkness of the printing of every page in the dictionary is different. This sometimes makes the input patterns difficult to recognize. Several locations must be tried to find the central position.

The purpose of searching for a circle pattern is to identify the stone patterns. A stone has a circle around it. This circle pattern is found by looking for an abrupt switch from 0 to 1 or from 1 to 0. Figure 2.5(a) shows the directions searched by PATTERN_GENERATOR to locate an abrupt switch from 0 to 1 or 1 to 0. If the initial points of the search are inside a letter pattern, the letter pattern itself will generate the 0-to-1 or 1-to-0 patterns. In this case, the search fails. Positions U and V are carefully selected so as not to be interfered with by the letter patterns, even if the center of the pattern is not precise. In the upper right-hand quadrant, the intersections U and V at the locations (3,8) and (4,8) relative to the center of the pattern are selected. From each point, PATTERN_GENERATOR searches horizontally and diagonally outward for the 1-to-0 or 0-to-1

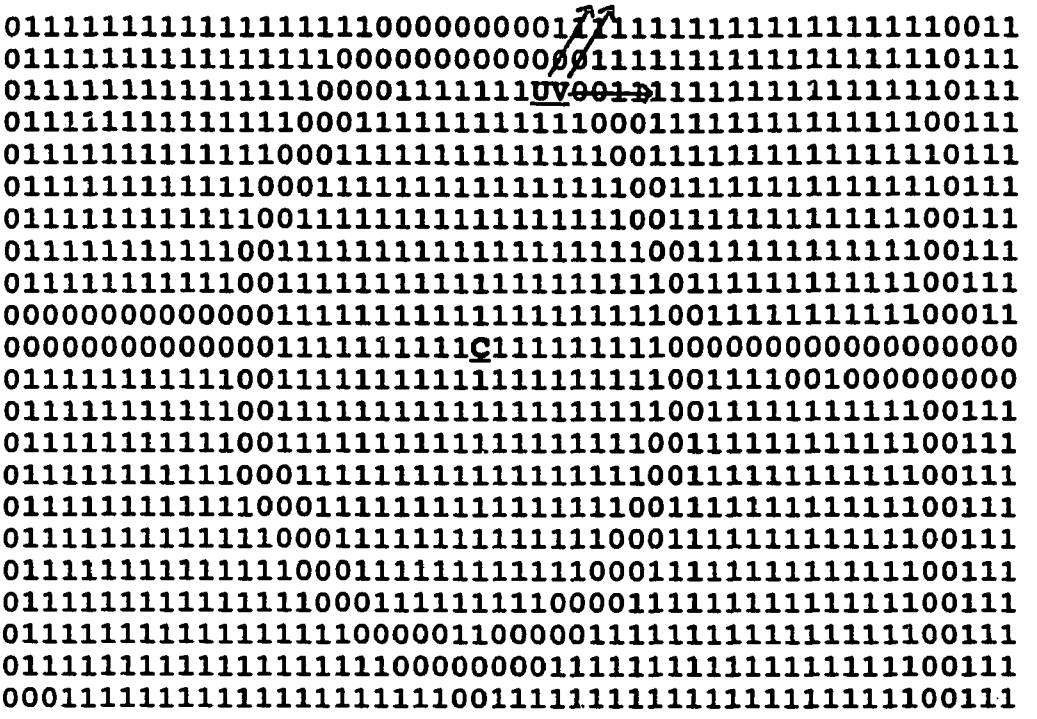


Figure 2.5(a) Directions to detect the circle pattern.

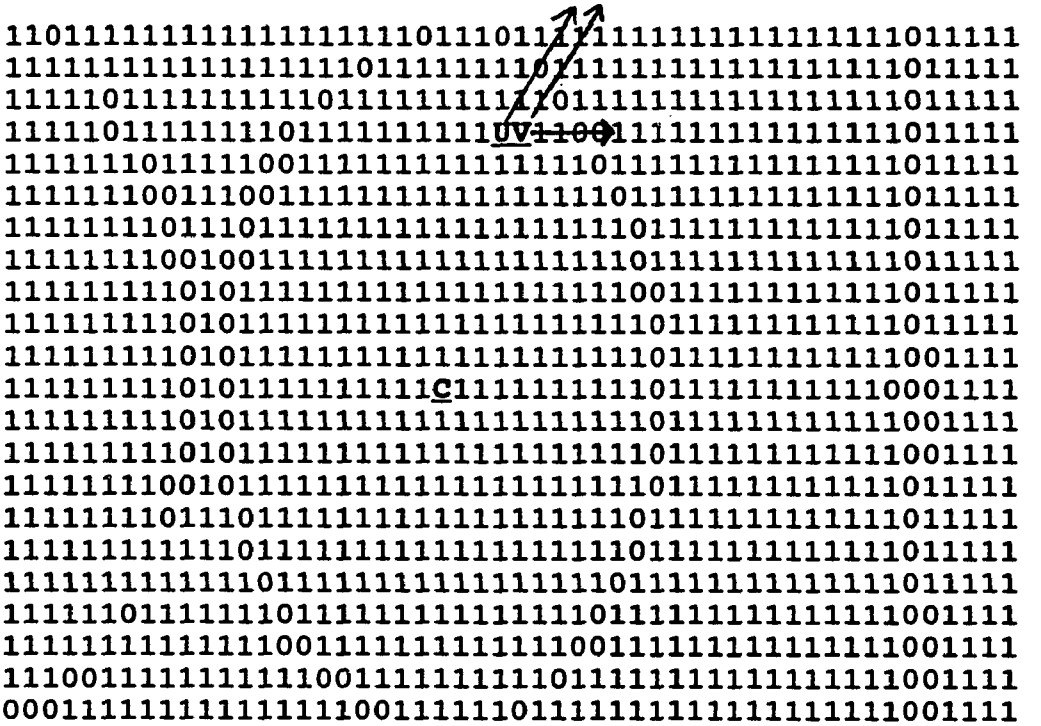


Figure 2.5(b) Failure to detect the circle may occur when the image is poor.

pattern.

The same search procedure is executed for the other three quadrants. The number-patterns in a stone sometimes generates an abrupt pattern during a search, but a circle pattern can still be found for these stones. If eight out of twelve searches succeed, a circle is presumed to exist.

The circle-pattern search method performs very well for a black stone. The discovery of a 0-to-1 pattern is bound to occur under all conditions, because of the pattern's simplicity. When the input pattern is difficult to recognize, the circle around a white stone becomes narrow. A simple diagonal search usually fails, as Figure 2.5(b) shows. The diagonal search from the point V goes through the circle without sensing any abrupt pattern. Adding an additional search, however, from point U increases the success rate of the entire search. If a circle exists and the searches from U and V both fail, the input image must be very difficult to recognize and the white stone's existence may elude the program.

The input pattern is then compared with a set of standard patterns stored in PATTERN_GENERATOR to recognize the stone and the number on the pattern. Figure 2.6 shows the standard pattern for the black stone "3" and white stone "3" cases. By

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 1 1 0 0 1 1 0 0 0 0
0 0 0 0 1 1 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 1 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 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 1 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 1 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 2.6(a) "3" pattern for BLACK.

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 1 1 1
1 1 1 0 0 0 0 0 0 0 0 0 1 1 1
1 1 1 0 0 0 0 1 1 0 0 0 0 1 1
1 1 1 0 0 0 1 1 0 0 0 0 0 1 1
1 1 1 1 1 1 1 0 0 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 1 1 1
1 1 1 1 1 1 1 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 0 0 0 0 1 1
1 1 1 0 0 0 1 1 1 0 0 0 0 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Figure 2.6(b) "3" pattern for WHITE.

identifying the standard pattern which has the least number of mismatching pixels when compared with the input pattern, the program knows which number is on the stone or which letter is on the location.

One common error generated by the input pattern is the rotation error. All horizontal lines of the input pattern are supposed to be represented as horizontal in the program. But, often, they are not. Such an error is introduced by the angle between the photo-sensitive device array in the scanner and the horizontal line in the joseki figure being scanned. The angle was carefully adjusted to make it as small as possible. On the other hand, PATTERN_ GENERATOR processes only subpatterns at each iteration. This makes the rotation error smaller and less sensitive. When the alignment is not good, however, the success rate of the algorithm is greatly decreased.

2.2.3 Verifying the Results of the Pattern Recognition

The second subprogram is built to interactively double-check the results of PATTERN_GENERATOR. The results from PATTERN_GENERATOR's double-checking subprogram are shown graphically in Figure 2.7. The routine allows a user to delete a stone, update a position, change the sequence number of a stone, etc. Two lines of numbers are shown on the

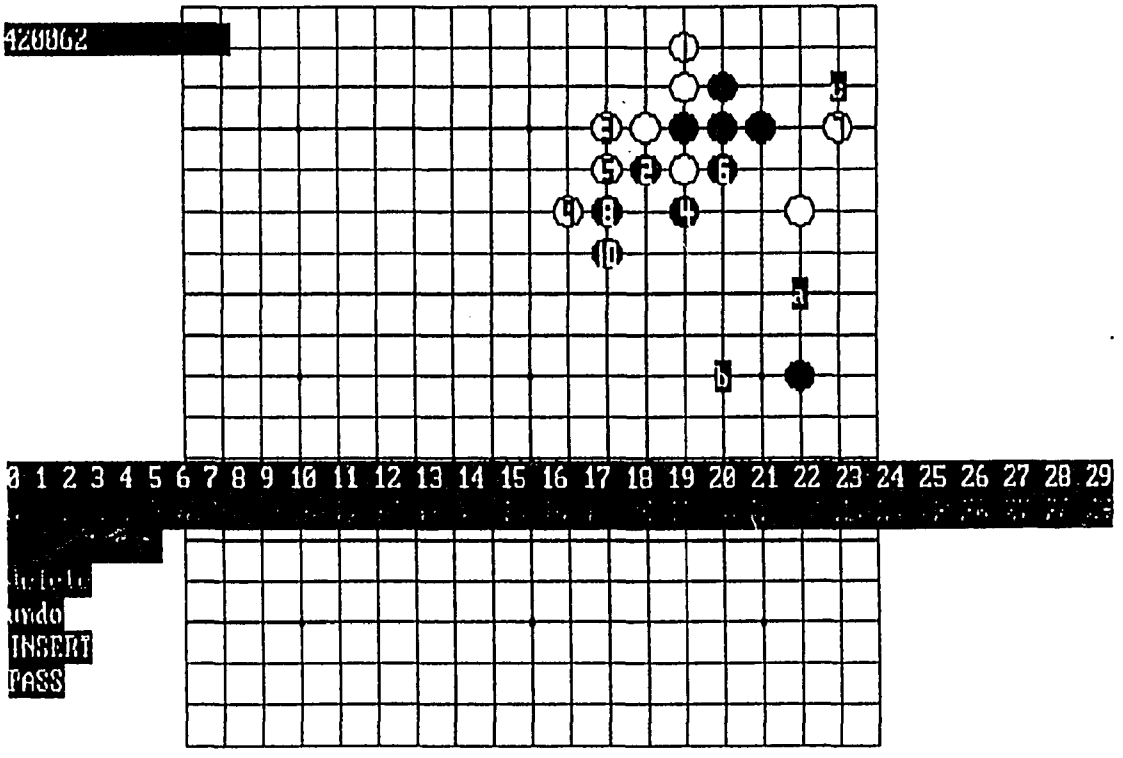


Figure 2.7 The display of the double-check program.

display. These two lines represent the stones and the numbers. The upper line is for the black stones and the lower one is for the white stones. Whenever there is an error, the mouse can be used to correct the situation. Likewise, the third line is for the correction of errors in the letters.

The program also checks the results automatically. The number of white stones and black stones in the base figure are checked. If the difference between the two numbers is more than one, the user is alerted. If the numbered stones are out of sequence, the user is also alerted.

2.3 Building the Joseki Tree

2.3.1 TREE_BUILDER

The information generated by PATTERN_GENERATOR is a set of records (x, y, color, sequence_number). "sequence_number" gives the sequence number of the stone the record represents. The records for the stones in the base figure have a sequence number of 0 and are called *base records*. They are grouped together before the other records that have a non-zero sequence number. Some joseki figures derive from an empty corner. The base records of those joseki figures are given a sequence number manually so that they can be derived from the

root of the joseki tree. The root of the joseki tree itself is a dummy node connecting groups of josekis that have different first stones in the joseki sequence.

TREE_BUILDER generates the joseki tree. If a group of records has no base records, TREE_BUILDER copies the information from the records to the tree nodes and inserts them into the joseki tree one -by-one from the root. The node containing the sequence number n+1 is the son of the node containing sequence number n. Some josekis may generate the same joseki pattern. Inserting those josekis into the joseki tree allows two different nodes defining the same joseki pattern in the tree, and are called *duplicated joseki patterns*. These nodes are kept in an array called a *duplication table* and adversely affect execution speed. Because the number of duplicated joseki patterns are small, however, it does not decrease the speed in most actual cases.

When TREE_BUILDER processes a group of records which have base records, it places the stones represented by the base records on the game board to construct the base pattern. The program then tries to find a joseki pattern which is the same as the base pattern from the current (partially constructed) joseki tree. If the program cannot find a joseki pattern for the base figure, either the joseki tree or the base record contains incorrect data. Sometimes, moreover, PATTERN_

GENERATOR misses a stone in a joseki figure because of a difficult-to-recognize input pattern. This joseki figure will be incorrectly stored in a path of the tree. The josekis which used this joseki figure as its base figure expect to find a joseki pattern from the joseki tree to match its base pattern. Because the joseki figure is stored incorrectly, TREE_BUILDER cannot find one of the joseki patterns which matches the base figure. Once TREE_BUILDER reports a failure, some records are modified manually. Lost stones can then be recovered.

2.3.2 Processing the Killing Condition

When we construct a joseki pattern by following a path in the joseki tree, some stones are killed and removed. When a program tries to reconstruct a joseki pattern, it has to check if some stones are killed when a new stone is put on the game board. Death-stones checking is a time consuming process, because the game board must be checked every time a new stone is placed on it.

This problem was resolved by pre-processing and keeping the result of the death-stones checking in a table termed a *killing table*. The structure of the table is defined by the following C code:

```
struct  killingtabletype {
                    int  killing_index;
                    int  killed_index;}  killing_table[50];
```

The node which represents a stone that kills some other stone(s) is called a *killing node* . The node that represents a stone which may be killed by some other stone is called a *be-killed node*. The integer variables killing_index and killed_index are the indices of the killing node and the be-killed node, respectively. If a node kills n nodes, there are n entries storing these indices.

The killing table is constructed by TREE_BUILDER. Every time a new node is inserted into the joseki tree, TREE_BUILDER also places the stone represented by the node onto the game board. It checks if there is any stone totally surrounded by the enemy links. The stones that are killed are removed from the game board and the indices of the killing node and killed node(s) are written into the table. TREE_BUILDER also marks the property field of a killing node. The be-killed nodes are found by searching the path backward from the killing node to the root and are then also marked. Finally, the nodes which have the same coordinates as the stones to be removed are marked.

A program that uses the joseki tree must access both the

joseki tree and the killing table. When the program traverses the joseki tree looking for joseki patterns, it checks the property field of the nodes it accesses. If a node is marked as a killing node, a killing takes place at the node. The program which uses the joseki tree does not go through the time-consuming death-stones detection.

Besides the joseki sequence, many figures are given for alternative moves deviating from the joseki sequence. These alternative moves often lead to unbalanced situations. The commentary in the dictionary then notes whether BLACK gets the advantage or WHITE gets the advantage. These non-joseki sequences are also inserted into our joseki tree. At the end of the sequences, we mark the last node "white_advantage" or "black_ advantage".

Figure 2.8 shows part of the joseki tree. "*" is the root. The numbers are the indices of the nodes. The letters beneath the numbers show part of the properties of the nodes. For example, "k" represents "be_killed" property, "K" represents "killing", "J" represents "major_joseki", (A major joseki is a joseki commonly used.) "j" represents "joseki", and "w" represents "white_ advantage".

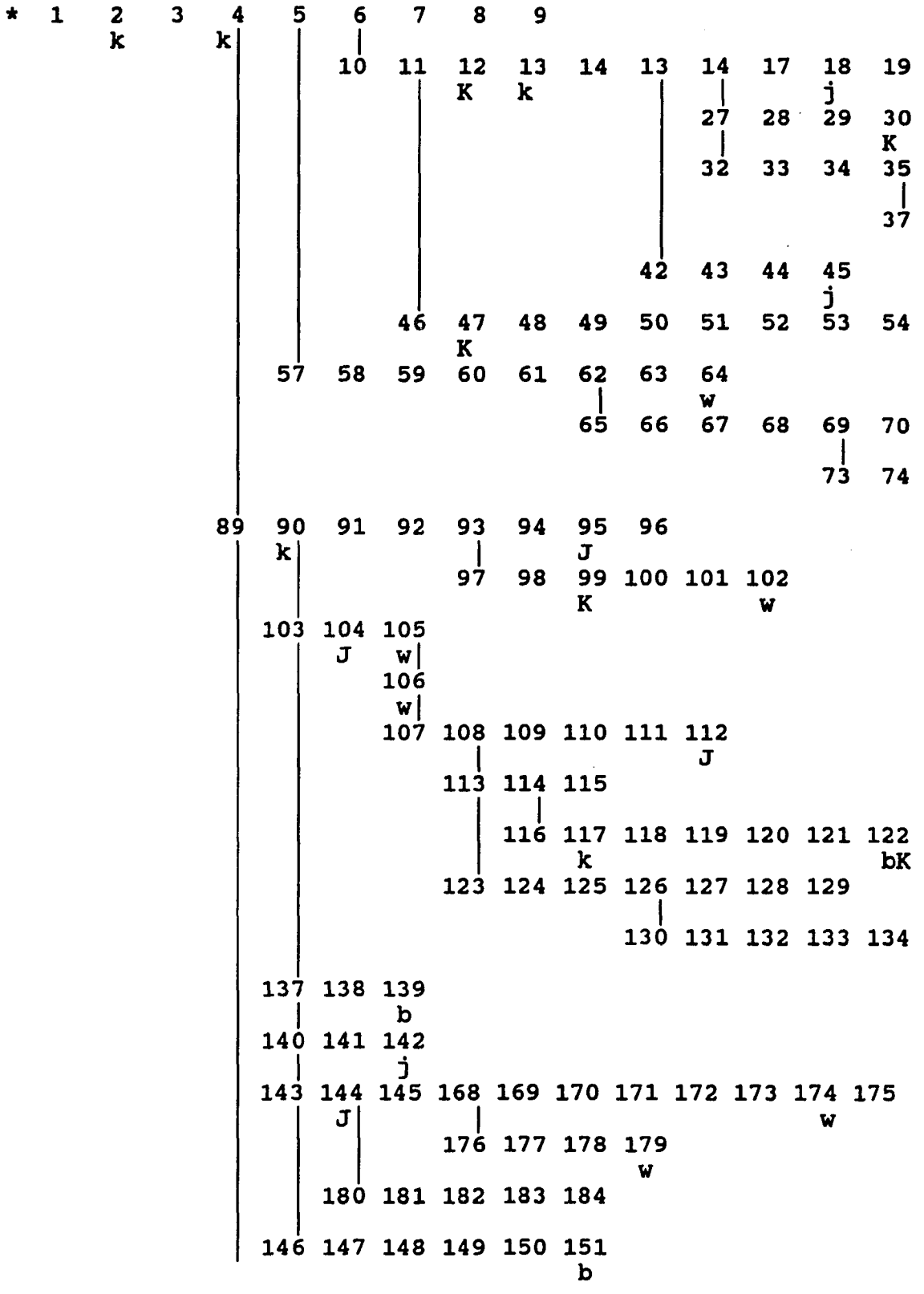


Figure 2.8 Part of the joseki tree.

CHAPTER 3

Discovering the Joseki Patterns

3.1 The Methodology

A *game pattern* is the pattern generated by the stones on the game board. If any game pattern's subpattern is a joseki pattern stored in the joseki tree, a node in the tree defines the joseki pattern. To find the node, a program named JOSEKI_FINDER walks through the entire joseki tree searching for the node.

JOSEKI_FINDER searches the path of the joseki tree starting from the root. It inspects the nodes at the next level to see if there is any node that matches a stone on the game board. For each matched node, JOSEKI_FINDER goes to the node and inspects the next level. The matching process is performed as deeply as possible along the path until a mismatch occurs. The last matching node defines the joseki pattern. The stones in the joseki pattern are a subset of the stones on the game board. The stones which are not in the subset become the interferences to the joseki pattern.

There are two kinds of mismatches. *Empty-mismatch* is a mismatch that occurs when the JOSEKI_FINDER tries to match a node with an empty intersection. None of the nodes in the joseki tree contain the "empty" color. When a mismatch occurs at a non-empty intersection, it is called *color-mismatch*. Empty-mismatch flags the end of the matching process. A joseki sequence can then continue from the empty intersection. The node that causes the empty-mismatch provides the next move for the joseki sequence. A color-mismatch is a real mismatch between the game pattern and the joseki sequence under inspection. Unless the node that results in the mismatch is removed, no joseki can be found from the node.

Sometimes, the above argument is not true. Figure 3.1 shows a path in the joseki tree and this path generates a joseki pattern. "*" is the root node. Nodes 1, 3 and 5 are the nodes that affect the intersection X in the matching process: node 1 creates a white stone at the intersection X; node 3 is a killing node removing the white stone; node 5 then creates a black stone at the now-empty position X. Position X begins at "empty" and is updated to color BLACK. The joseki should have a black stone at location X. Assume that the joseki pattern is now on the board. JOSEKI_FINDER should have a match between the game pattern and the path. But, it generates a mismatch and stops searching the path when

position X matches with node 1 before reaching node 3. JOSEKI_FINDER fails to identify the joseki pattern.

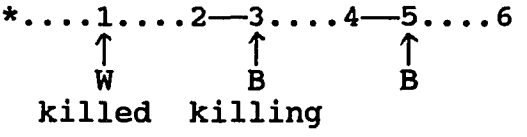


Figure 3.1 A hard-to-identify joseki pattern.

During the matching process, some stones die and are removed from the game board. Other stones are then placed in these now-empty intersections later in the sequence. The stones in these intersections are dynamically updated along the joseki sequence. These intersections are termed *dynamic intersections*. When JOSEKI_FINDER searches the front part of the path and a mismatch is indicated by the program at one of the dynamic intersections, JOSEKI_FINDER cannot be sure that a mismatch has really occurred. The stone at the position may be updated later when the rear part of the path is checked and finally matches the stone on the game board.

3.2 Processing Dynamic Intersections

The matching procedure was modified for dynamic intersections. JOSEKI_FINDER does not match the intersections where the stone may be killed in a joseki sequence until an

empty-mismatch occurs. An array called "history" is used to store the histories of the color for these dynamic intersections. The "history" array is defined as follows in C:

```
struct histype {
                int x, y;
                int color;
                int index; } history[20];
```

(x,y) are the coordinates for a dynamic intersection. (x,y) is called the *position in the entry* and the entry *points* to the intersection (x,y). The integer variable "color" stores the history of the colors for the intersection. It is called the *color in the entry*. The integer "index" helps JOSEKI_FINDER find the last node which updates the entry.

When JOSEKI_FINDER matches a node with the stones on the game board, it checks the entries in the history array. If the position in the node is not in the history array, JOSEKI_FINDER checks the property field of the node to see if the node is a be-killed node. A be-killed node, recall, represents a stone that may be killed by another stone later in a joseki sequence. JOSEKI_FINDER copies the (x,y) coordinates and the color in the node into an empty entry of

the history array if the node is a be-killed node.

If the (x,y) coordinate in the node can be found in one of the entries in the history array and the color in the entry is "empty", the stone at the intersection must have been removed by some other stones. A new stone can then be placed at the location. The color in the entry is updated to the color in the node. When the position in the node is in the array and the color in the entry is not "empty", an error is generated if a stone tries to update the color in the entry.

When a stone generates a killing at the position pointed to by the entry, the following rules apply. If a node being matched has the property "killing", a killing takes place at the node. JOSEKI_FINDER finds all the nodes that are killed by the stone from the killing table. For each node killed, the program checks through the entries in the history array. If the killed position is not in the array or the color is "empty", there is an error. Otherwise, the color in the position is updated to "empty". The color in the entry can be updated to BLACK or WHITE later.

After an empty-mismatch is found, JOSEKI_FINDER compares all the entries in the history array with the stones at the dynamic intersections. If all of them are matched, a joseki pattern is found.

The "color" field in each entry of the history array is used as a stack to keep the history of the color in the entry. A color is represented by two bits for BLACK, WHITE, and "empty". The first two bits, bit 0 and bit 1, show the current color. To update the color, the "color" integer variable is shifted left two bits and the new color is added to bit 0 and bit 1. Shifting the "color" integer right two bits restores the previous color. The color history can push up to eight colors onto the stack. This is enough for the joseki search. When there is no color in the color stack, the entry is deleted.

Not all dynamic intersections in the array change their colors during a matching process. For example, Figure 3.2 shows part of a joseki tree. Path 1 contains the nodes from the root to node 1 and node 2. Path 2 contains the nodes from the root to node 1 and node 3. Node 3 kills node 1. Node 1 is labeled as a be-killed node. Node 1 is not killed by any node in path 1. When the matching algorithm processes path 1, node 1 causes the insertion of a new entry to the history array because it is a be-killed node. The entry does not perform any updating operation on the color field when path 1 is searched. The intersection is checked after an empty-mismatch is found in path 1.

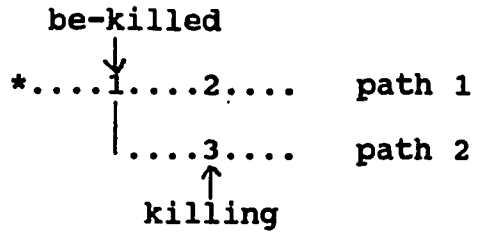


Figure 3.2 A be-killed node is shared by two paths.

3.3 The History Array and Its Matchings

A matching point is a node that forms a path to the root along which the stones on the board are matched. Before JOSEKI_FINDER processes the history array, a matching point is assumed to exist at the node right before an empty-mismatch. When there is a mismatch between the game pattern and the history array, the matching point is modified.

In Figure 3.3, node 6 generates an empty-mismatching and all nodes except those processed by the entry for some intersection X have been matched. Node 5 causes the program to place a black stone at intersection X. The entry in the history array processing position X has the color BLACK. Assume that intersection X is now empty. Matching the stone on the game board and the history array generates a mismatch at the entry. At node 4, the color at position X is empty, because node 3 generates a killing at the intersection. After node 5, the color at position X changes to BLACK. The

matching point should then be at node 4. However, if the stone on the position X is a white stone, the matching point should be at node 2.

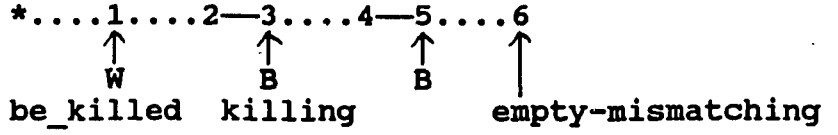


Figure 3.3 An example of adjustment of the matching point.

The following 2 steps modify the algorithm to find the matching point when a mismatch occurs between the history array and the game board at position X:

(1) Trace the path from the root to the empty-mismatching point. Find the last node which set the color for position X on the game board to match the color in the entry for position X in the history array. Several nodes may set the color in position X. The node which most recently set the color of the entry is selected.

(2) Trace the path from the node to the empty-mismatching point until the node Z which updates position X to another color is found. Then, the parent node of node Z is the matching point.

In Figure 3.3, if a white stone is at position X, the algorithm first finds the node 1. Node 1 is the last node that sets the position X to WHITE. It then finds node 3 by step (2). Node 3 is the next node which changes the color at position X. Node 2 is the matching point.

For each mismatch entry, JOSEKI_FINDER finds a matching point using the above steps. The matching point which is closest to the root is the overall matching point.

Figure 3.4 shows another example of the recognition of a joseki pattern by the program. In it, node 2 kills node 1. Intersection X pointed to by node 1 is empty after node 2 is processed. On the game board, the game pattern can be created by placing a black stone at the intersection pointed to by node 2 and putting nothing at X. A killing event does not necessarily happen on the game board although it is indicated as such in the joseki sequence. This is because position X may be empty to begin with, rather than empty only after a killing. Procedure 3.1 shows the algorithm to process a node.

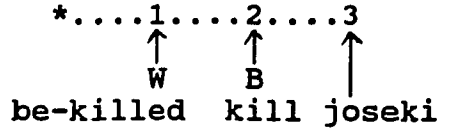


Figure 3.4 Recognition of a joseki pattern under a killing situation.

```
MATCH_A_NODE(Node)
{
  get (x,y) which is the position in Node;

  if (x,y) is in one of the entries in the history array,
       update the entry;

  else if Node is a be-killed node,
       generate one entry in the history array for Node;

  else if the color of the stone at (x,y) on the game board is
            different from that in Node,  /* mismatched */
          {
           restore history array;
           return;
          }

  /* now Node is either in history array or a match occurs */
  if Node is a killing node,
     {
       find all the nodes killed by Node;
       for each such node, /* they must now be in history
                              array */
           update the color of the entry in history array
           pointed by the node to "empty" and save the killed
           stones for restoring;
      }

  for each child node, ChildNode, of Node,
       MATCH_A_NODE(ChildNode);

  if all ChildNodes fail to match,
      {
        find the matching point by matching the history array
        and then adjusting the matching point to its new
        position;

        if there is no interference to the matched stones,
            {
              using the matching point as the root,
              perform alpha-beta pruning and influence
              evaluation;

              if the result is better than previous results,
              save the node leading to the result;
             }
      }

  restore history array (including killed stones);
}
```

Procedure 3.1 The algorithm which processes a node.

3.4 Resolve the Symmetric Problem

One joseki has 16 equivalents. Figure 3.5 shows some equivalents. The game pattern in Figure 3.5(a) is a joseki. The mirror images of the joseki relative to the lines x=10, y=10, x=y, x=-y and the point (10,10) are also josekis. Switching the colors of the stones in a joseki also creates a joseki. 3.5(c) are two equivalents of the joseki shown in Figure 3.5(a). Storing all the equivalents of a joseki into a database wastes disk space and decreases execution speed. Chen [5] suggests that one store a joseki but generate its equivalents to match the Go pattern on the board. JOSEKI_FINDER, in contrast, solves the problem by using 16 game boards, one for each of the 16 equivalents. These game boards are generated by the application of 16 conversion functions, shown in Table 3.1.

In Table 3.1, xs and ys are the x and y coordinates for the new game board. cs is the color after conversion. c' is BLACK if c is WHITE and WHITE if c is BLACK. Each conversion function creates a new game board. JOSEKI_FINDER matches the stones at these 16 boards' upper right-hand corners only. When there is a match, the location of the next move (shown in Figure 3.5 as a grayed stone) is found first. Then, it is reconverted to the position on the original board.

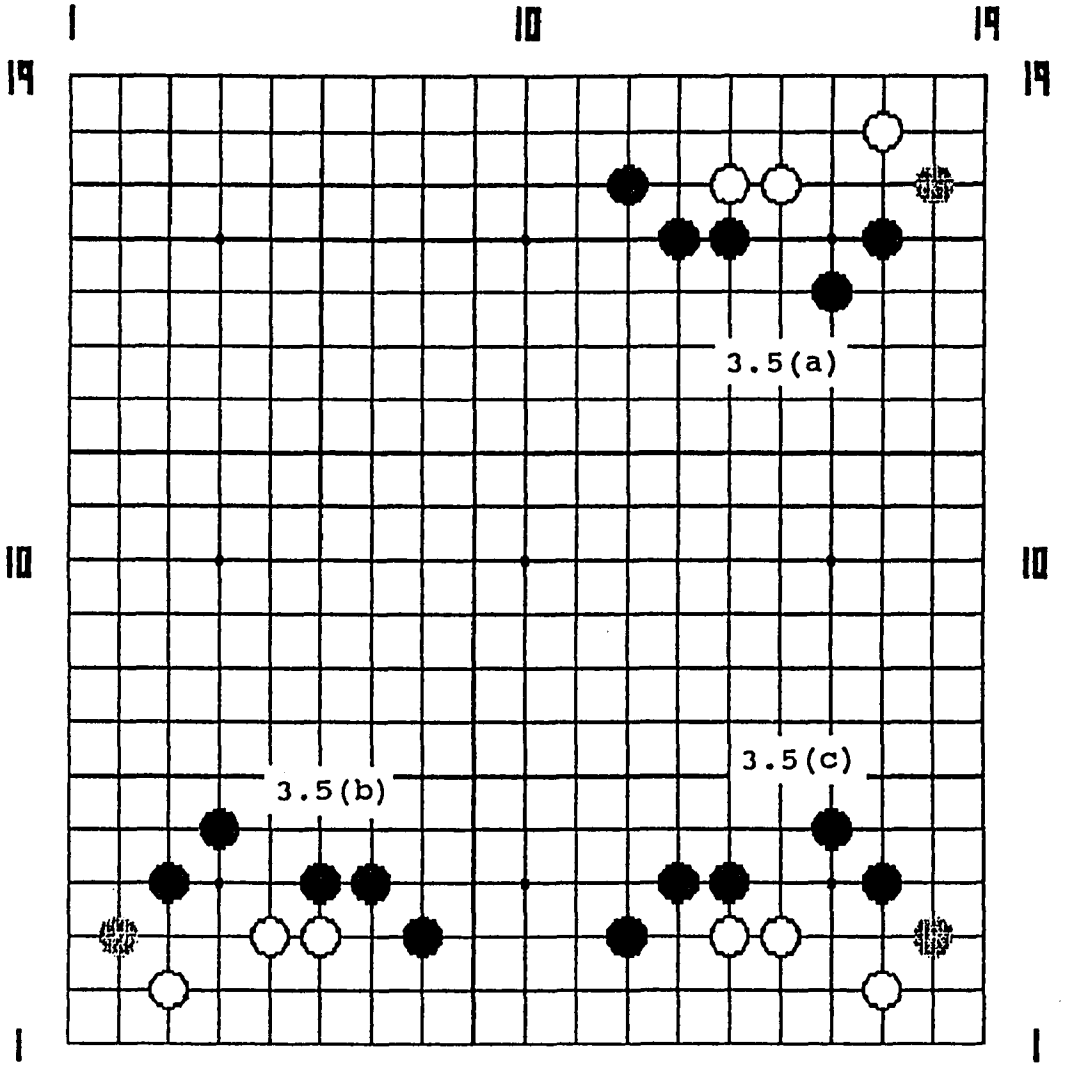


Figure 3.5 3.5(b) and 3.5(c) show two equivalents of the joseki shown in 3.5(a).

board	conversion function		
0	xs=x	ys=y	cs=c
1	xs=x	ys=y	cs=c'
2	xs=x	ys=20-y	cs=c
3	xs=x	ys=20-y	cs=c'
4	xs=20-x	ys=y	cs=c
5	xs=20-x	ys=y	cs=c'
6	xs=20-x	ys=20-y	cs=c
7	xs=20-x	ys=20-y	cs=c'
8	xs=y	ys=x	cs=c
9	xs=y	ys=x	cs=c'
10	xs=y	ys=20-x	cs=c
11	xs=y	ys=20-x	cs=c'
12	xs=20-y	ys=x	cs=c
13	xs=20-y	ys=x	cs=c'
14	xs=20-y	ys=20-x	cs=c
15	xs=20-y	ys=20-x	cs=c'

Table 3.1 Conversion functions for the generation of equivalents.

The boards 1, 3, 5, 7, 9, 11, 13, 15 are generated by switching the colors of the stones on the boards 0, 2, 4, 6, 8, 10, 12, 14. Board 0 is the original game board. Board 2 is the result of performing a reflection over the line y=10. It converts the lower right-hand corner to the upper right-hand corner. Board 4 performs a reflection over the line x=10. It converts the upper left-hand corner to the upper right-hand corner. Board 6 is the result of performing a reflection around the point (10,10). It converts the lower left-hand corner to the upper right-hand corner. Boards 8, 10, 12, 14 result from reflections of the line x=y from the boards 0, 2, 4, 6. In Figure 3.5(b) and 3.5(c), we show, as examples, boards 6 and 2, respectively.

3.5 Processing the Empty Corners

When stones are placed on the game board, the joseki sequence which generates a color-mismatch becomes invalid. All child nodes of the mismatching node are skipped during the match process. When more stones are placed in a corner, the number of invalid josekis increases because the chances of having a color-mismatch increase.

An empty corner takes a long time to process because there is no invalid joseki sequence and all the josekis must therefore be tested in order to find the best move. In order

to reduce the time it takes to process an empty corner, 60 opening games are collected to help the program find the next move at the empty corners quickly. The moves of these opening games are not restricted to a corner. Figure 3.6 shows one of the opening games stored in the program, where the players place the stones in four corners. Those 60 opening games are stored in a tree like an ordinary joseki tree. Each opening sequence also has 16 equivalents; our program processes them all. The longest sequences in the opening-game tree are currently five.

JOSEKI_FINDER applies the matching algorithm to the opening- game tree as it does in matching a joseki tree. However, no "be-killed" and "killing" nodes are in the opening tree. The "property" field in the nodes is ignored. A matching point in the opening-game tree may have several candidate next-moves to select from the tree. JOSEKI_FINDER selects the first child node of the matching point as the next move. Since these moves are played by professional players, they are all favored moves.

If the opponent is known to be weak in certain types of opening games, or a specific strategy is decided on, we may adjust the tree by placing the favored move in the first child of a node.

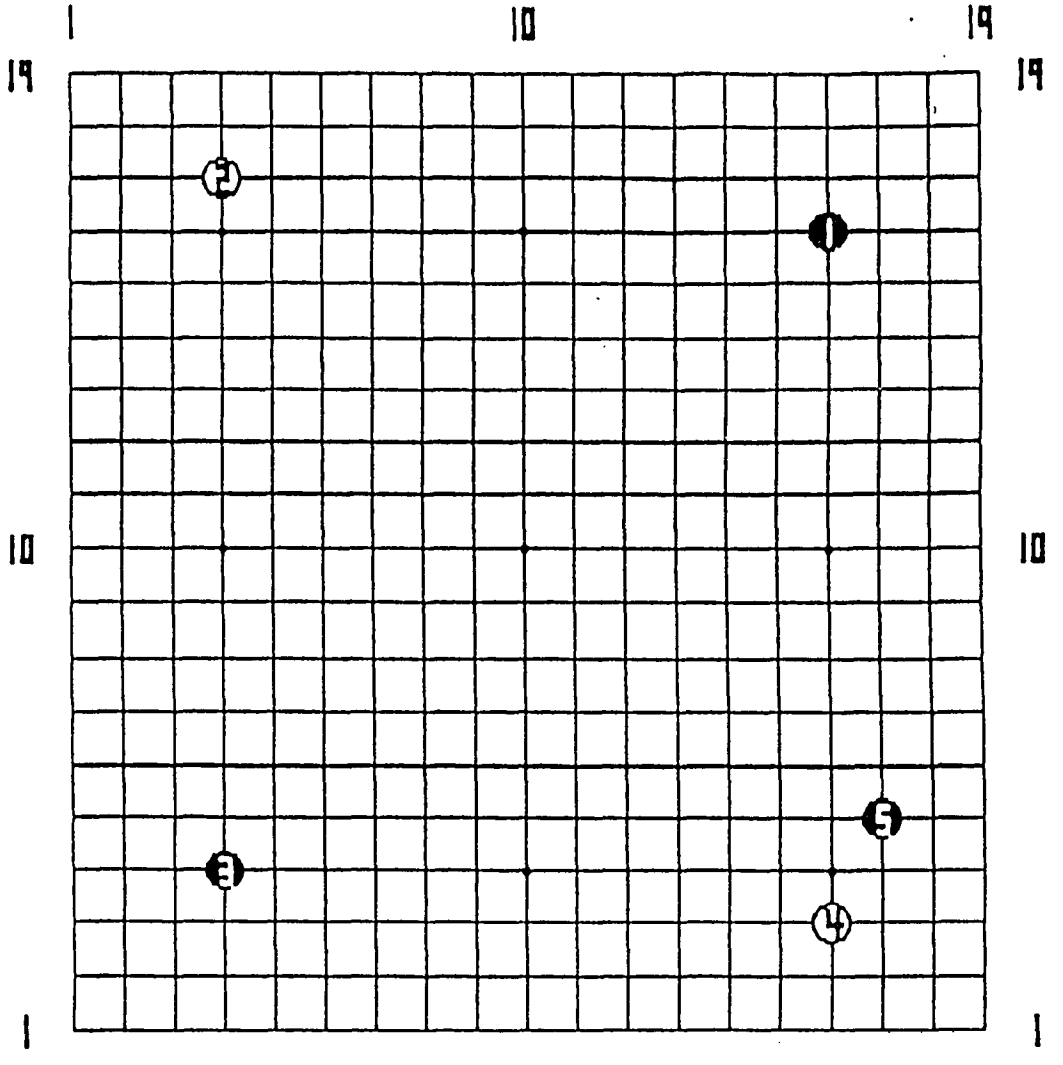


Figure 3.6 One of the sixty opening games stored in our program.

The number of opening games can be extended to increase the possibility of matching. If no matching point is found for an empty corner, the program selects a default position without evaluating all possible josekis. The default position we prefer is at (17,16) and its equivalents. Starting from this position, there are plenty of josekis to choose from.

However, it is better to add a set of rules to help decide which move to select without using a default move. This will generate much better results in selecting a move for an empty corner. Of course, these rules depend heavily on the strategy and must be adjusted before each game. These strategies can be embedded in the rules.

3.6 Loading the Joseki Tree Dynamically

3.6.1 Database Segmentation

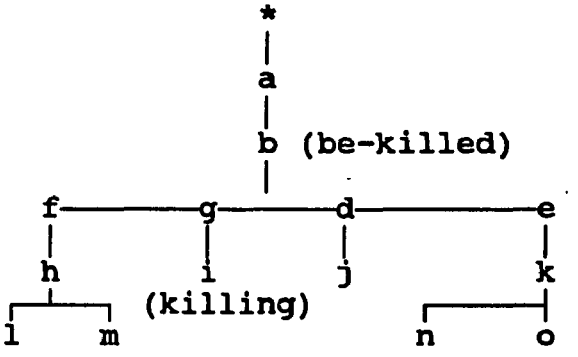
The entire joseki tree is stored on a disk. Accessing the tree requires loading the tree from the disk which is time-consuming. Since only some of the josekis are used during a game, to increase efficiency it is only necessary to load the subtrees which contains the valid josekis, not the entire tree.

The entire joseki tree is split into 30 subtrees. A subtree is loaded only if it is necessary. Each subtree is an independent unit. The tree X in Figure 3.7(a) is split into two trees, A and B, as shown in Figure 3.7(b). As can be seen, matching tree X gives the same result as matching tree A and tree B. There are some duplications among the trees. Nodes a, b must be replicated in both tree A and tree B. The positions and the colors in the nodes a, b are the same in both tree A and B. The "property" of the nodes, however, may be different. In tree A, node i kills node b. Node b in tree A has the property "be_killed". In tree B, no node kills node b.

3.6.2 Discovering Josekis in a Dynamic Space

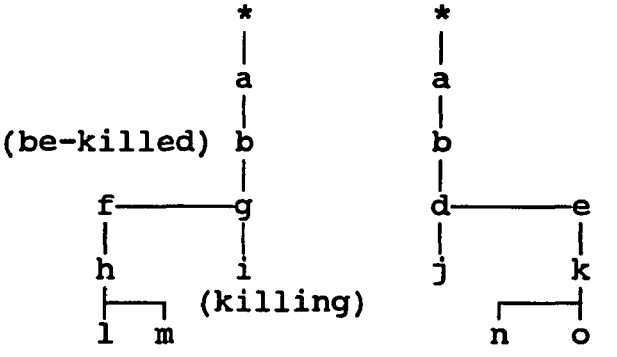
The matching algorithm we developed can find a joseki from a non-joseki sequence. If an algorithm can find the joseki patterns which might be generated from a non-joseki pattern, such an algorithm will load only those joseki subtrees which contains useful joseki patterns. Then, those subtrees alone will be checked.

To make the dynamic algorithm work, we duplicate and save part of the joseki tree in memory throughout the matching process. We copy the nodes in the upper four levels in every



Tree X

3.7(a)



tree A tree B

3.7(b)

Figure 3.7 Splitting tree X into two subtrees.

subtree and merge them into a single tree. This tree is called a *cap tree*.

From the experiments, most of the mismatches occur from level 1 through level 4 in the joseki tree. Even when a subtree is loaded, most nodes are never used because the matching process stops after a mismatch (typically) occurs close to the root. Usually, having the cap tree in memory has the same effect as having the entire joseki tree residing in memory. A table is used to map a node in the cap tree to the node in the joseki subtree. For every node in the tree, we can find out which tree it belongs to by using the table.

When there is no empty corner, the program matches the nodes in the cap tree with the stones in a game pattern. When there is a match in the cap tree and no interference to the match case, the subtree that contains the matching node is loaded. The subtree name is then registered. A further search in the same corner or another corner may result in a request to load the same subtree again. The registration of the subtree avoids reloading the same subtree.

Because searching for a pattern from a joseki tree in memory is much faster than searches through secondary storage, JOSEKI_FINDER matches the tree with the patterns at each of the four corners whenever the pattern of a corner causes a

tree to be loaded into memory.

If a matching process matches the node in level 4 (lowest level in cap tree), the subtree is loaded without checking the interference. JOSEKI_FINDER cannot determine whether a stone which is not being matched is an interference, or whether it will be matched after the subtree we selected is loaded. Because the four stones which cause the match in the cap tree always exist unless some are killed, the subtree must be loaded every time the program searches for the next move. The program can save the tree in memory to avoid reloading it from disk if there is enough memory. The program can also be extended to save the branches of the subtree in memory and attach it to the cap tree. The branches then become part of the resident tree. When the cap tree is searched, these branches are also searched.

The number of nodes in the cap tree is small. In our database, only 391 nodes are in the cap tree.

3.7 Evaluation

3.7.1 Invalidation of a Joseki Pattern

A joseki normally is described as occurring in an empty

corner with no other stones in the nearby area. A stone in the nearby area changes the utility of the joseki. After a joseki pattern is found, the stones surrounding the pattern become the interferences to the joseki pattern. These stones might invalidate the pattern.

In JOSEKI_FINDER, each stone generates an interference area as shown in Figure 3.8. (The intersections marked '1' are the interference area generated by the stone at the center.) For a game pattern, the stones in the pattern generate a large interference area surrounding the pattern. Figure 3.9 shows the interference area generated by a game pattern. The area marked by the small rectangles is the interference area. Within this area, if there is any stone not belonging to the pattern, it is regarded as having interference on the pattern. All the josekis derived from the pattern are invalidated and this pattern is then given up immediately.

After a joseki pattern is found valid, alpha-beta search is used to find the next move. The matching point found by JOSEKI_FINDER becomes the root of the alpha-beta search tree. All the josekis derived from the matching point are evaluated. When all of the stones in the joseki that is currently under evaluation are placed on the board, they generate an interference area too. Some josekis under evaluation may be

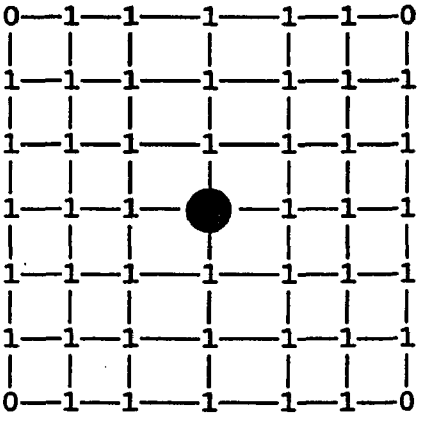


Figure 3.8 Interferences generated by a stone.

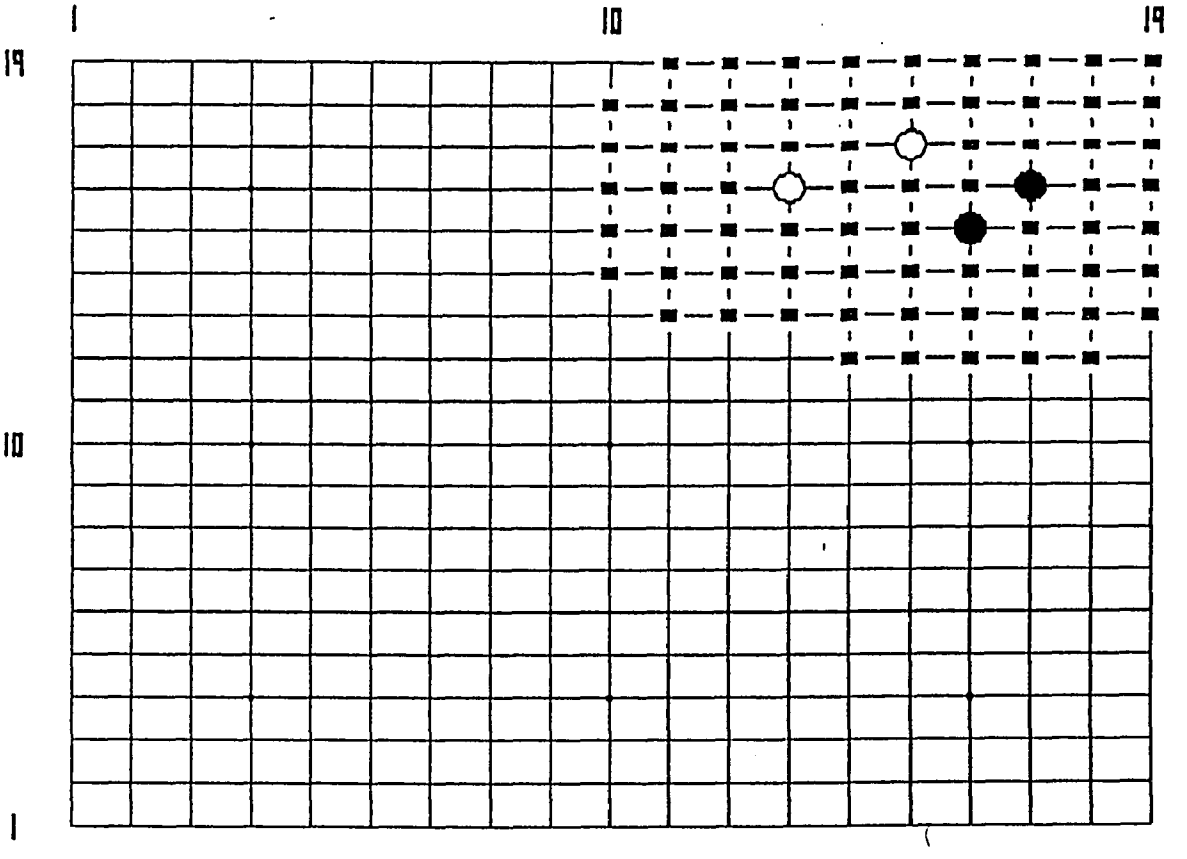


Figure 3.9 The interferences generated by a group of stones.

invalidated by interferences. If there is any stone in the interference area which does not belong to the joseki, the joseki will not be considered.

3.7.2 Alpha-beta Pruning

Deciding which joseki is a good one depends on the global situation. The alpha-beta search only evaluates the paths in the joseki tree. When it finds a node which is marked "joseki" or "major_joseki", it applies the influence evaluation function on the joseki.

A stone in the corner or border area shows a stronger degree of control toward the border area than does a stone in the central area over its surrounding area. For example, black stones at (10, 10) and (10,2) generate the influence 92 at (10,9) and (10,1), respectively. Both stones show the same degree of control at the locations as measured by the influence. But placing a white stone at (10,1) is more dangerous than placing it at (10,9). To reflect this fact, we modified the regular influence function by adding some "virtual stones".

Virtual stones stay outside of the game board at the lines x=0, x=20, y=0, y=20. They generate influence on the game board. For every intersection at the boundaries, we search

three intersections normal to the boundary looking for the closest stone to the boundary. If a stone is found, a virtual stone with the same color is inserted right next to the boundary, but outside the game board.

For example, assume that a game pattern has a black stone at (3,4) and a white stone at (2,4). The procedure starts from (1,4) searching the intersections (1,4), (2,4), and (3,4). When the white stone at (2,4) is found, a virtual white stone is added to the location (0,4). This virtual stone generates the same influences as the original stone. Stones three or more intersections away from the boundary are not supplemented by virtual stones.

A corner has two boundaries. A stone at the corner may be intensified twice by the two virtual stones generated by these boundaries. The influences generated by a stone at the corners are stronger than that of a stone in the border area.

In the opening game, the players try to outline a control area; they try to cover a large area with a small number of stones. The influence in an area should be moderate; not too strong, not too weak. An area with small influence is too weak to defend. An area that has a strong influence has a high density of friendly stones, which should be spread out so as to occupy more territories. The number of empty

intersections in various intervals (of influences) is weighted and summed. The joseki which has the highest weighted sum is selected.

The joseki search turns out to be fast. Along a path, if a mismatch occurs, the joseki pattern defined by the matching node is invalidated immediately. Most paths are quickly abandoned. For the valid joseki patterns, alpha-beta search applies the influence function to the valid josekis derived from the matching nodes to select a good joseki. Procedure 3.2 shows the algorithm of JOSEKI_FINDER.

3.8 The Performance of JOSEKI_FINDER

Figure 3.10 and Figure 3.11 show the performance of JOSEKI_FINDER on two examples. In these examples, JOSEKI_FINDER is tested by giving a game pattern at one corner, and it is directed to find a joseki at another corner.

In Figure 3.10, a group of stones is given at the upper left-hand corner. One black stone at (17,16) and one white stone at (15,17) are also given. Using JOSEKI_FINDER, BLACK finds the joseki at the upper right-hand corner from the joseki tree. The joseki suppresses the white stones into the corner and lets the black stones cooperate with the stones at the left-hand side enclosing the upper border. BLACK also has

```
FIND_A_MOVE()

{

  if number of moves so far is less than 6,
        {
        search opening-game tree;
        if a move is found,
             play the move and exit;
        }

  if there is an empty corner,
          play default move and exit;

  for each of the 16 equivalent game boards,
      {
         MATCH_A_NODE(root node of cap tree);
          /* matching the game board with cap tree;*/

          LEVEL=level of the matching point;

          if LEVEL=0, search the next game board;

          if (LEVEL<=3 and no interference to the nodes
              matched) or LEVEL=4,
            {
              SUBTREE = The subtree containing the matching
                        point;
              if SUBTREE was never loaded in this move-
              search,
                   {
                     Load SUBTREE;

                     for each of the 16 equivalent game boards
                        MATCH_A_NODE(root node of SUBTREE);
                    /* When a subtree is loaded, try to match
                       it with all 16 equivalent game boards
                     */
                    }
            }
    }
  if no move is found,  report "cannot find a move";
}
```

Procedure 3.2 The algorithm of JOSEKI_FINDER.

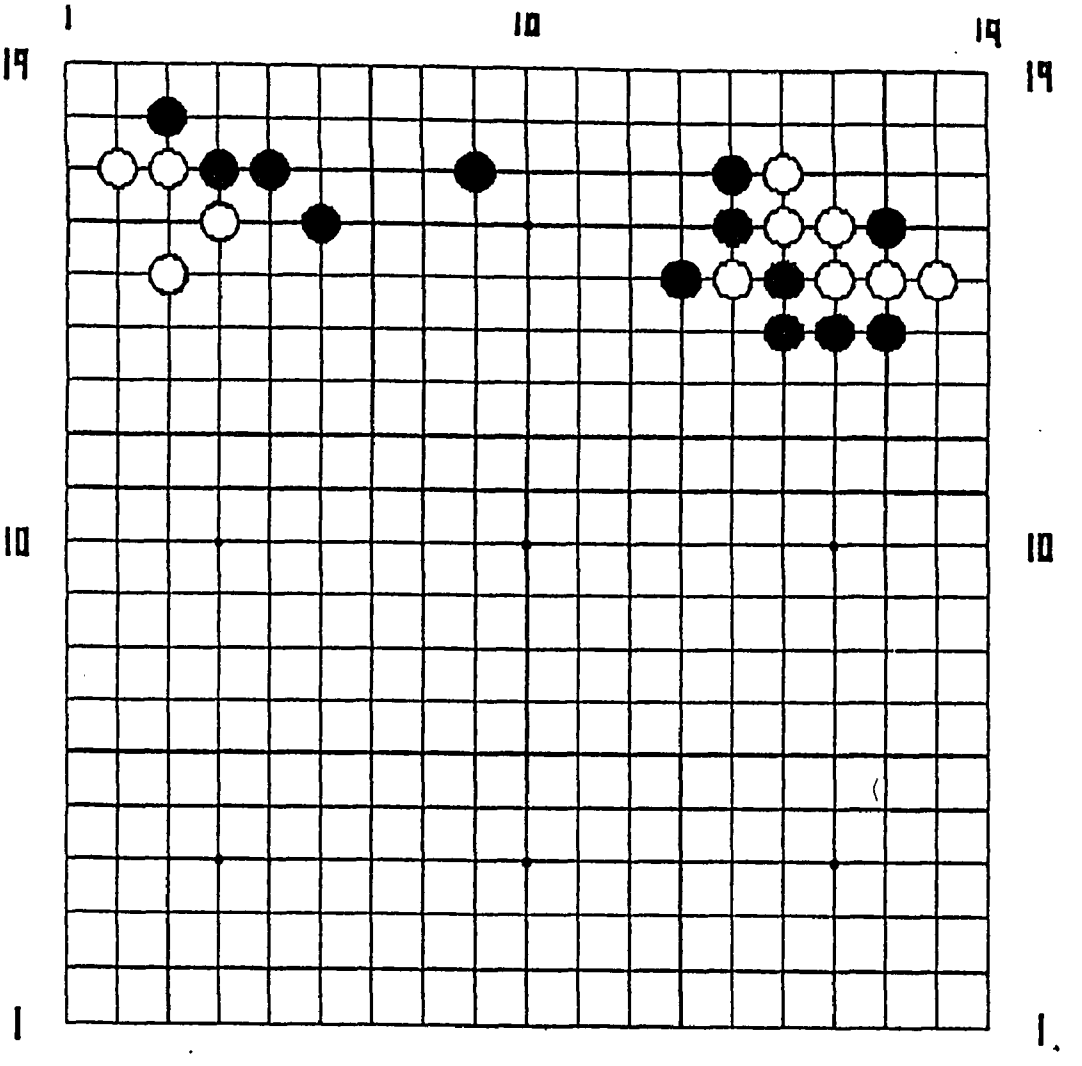


Figure 3.10 JOSEKI_FINDER selects the joseki at the upper right-hand corner for BLACK.

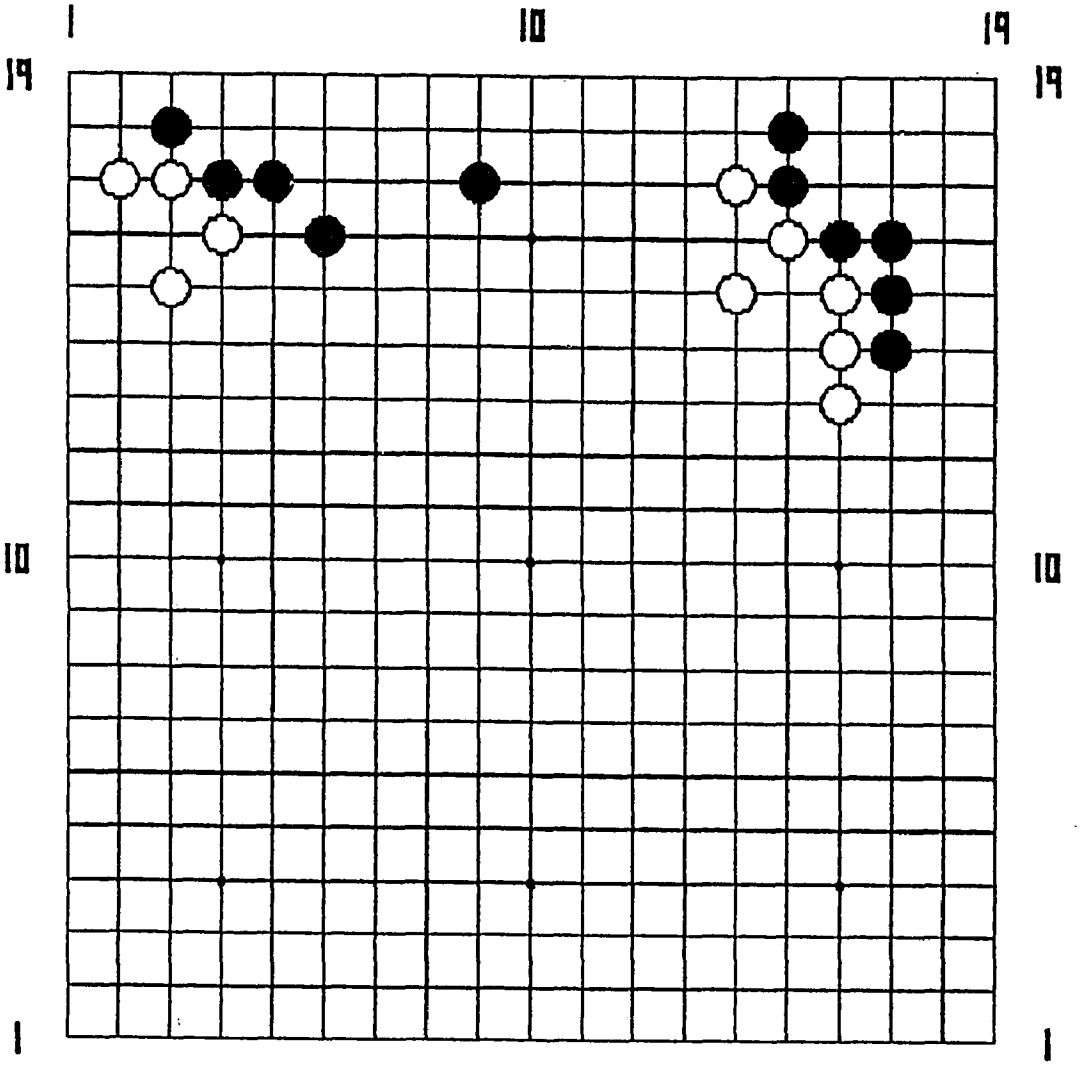


Figure 3.11 JOSEKI_FINDER selects the joseki at the upper right-hand corner for WHITE.

a better position facing the central area.

In Figure 3.11, the same pattern as that in the upper right-hand corner of Figure 3.10 is given. A black stone is placed at (17,16). WHITE has the next move. JOSEKI_FINDER selects another joseki which tries to interrupt the connection between the black stones in the right-hand side and the left-hand side. The white stones are safe and can be extended to the upper border area and the right-hand border area. In order to show the results clearly, we directed JOSEKI_FINDER to find a joseki only at the upper right-hand corner and to disregard the two empty corners at the lower border area in both examples.

The above two results show that JOSEKI_FINDER, given a global situation, is capable of finding a good joseki. Playing the josekis is no longer restricted to local games. By using the new influence function we created and the virtual stones, a good joseki can be found which co-operates with other groups of stones to create a better position.

During the development of JOSEKI_FINDER, hashing methods were tried. Hashing methods require two databases: one for the joseki tree and one for the hash table. It requires more complicated processing than does JOSEKI_FINDER and it frequently needs to load a tree which is not relevant to the

joseki that we want. Hence, we decided to drop its use.

After years of improvement, the best result we can get is using the cap tree to find a partially matching pattern. From there, the subtrees are selected for loading into memory and matching with the patterns at the four corners. The program then eliminates all the invalid josekis and performs an alpha-beta evaluation to select the appropriate joseki.

The positions (15,17), (15, 16), (16, 16), (17, 17), (17, 16) and their equivalents are the possible first stones of josekis given in the joseki dictionary [16]. The nodes representing those stones are the roots of many of our joseki subtrees. When only one stone is in each corner and it is in one of those positions, many subtrees must be loaded for matchings and evaluations. There are fifteen subtrees in which the first stone starts at (17,16). When there is a single stone at (17,16), JOSEKI_FINDER loads the fifteen subtrees for matchings and evaluations. If four single stones at four different corners are as shown in Figure 3.12, almost all 30 subtrees must be loaded to find the appropriate joseki. Parallel processing can help resolve this problem.

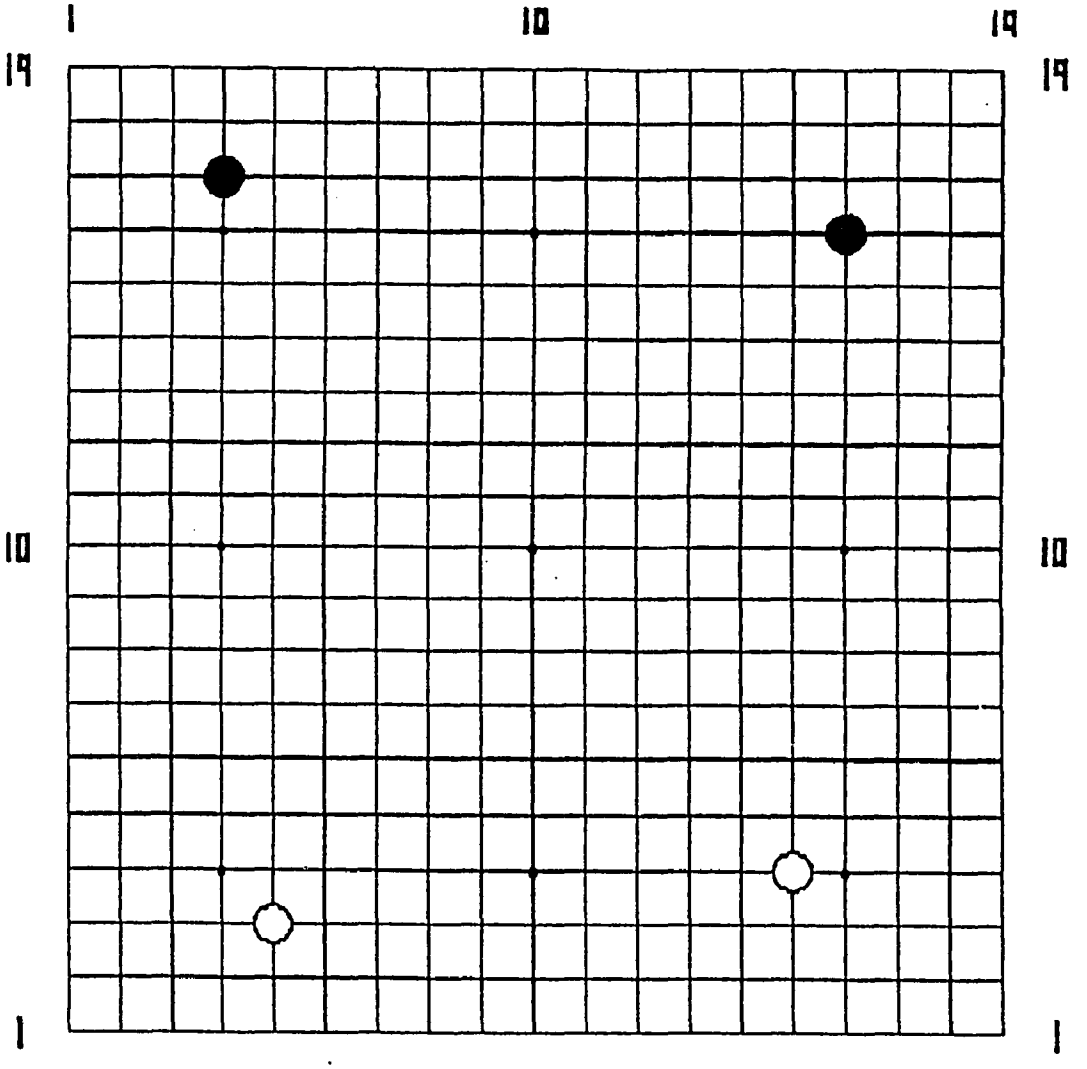


Figure 3.12 Four different first-stone positions in the joseki dictionary.

CHAPTER 4

Parallel Processing

4.1 The iPSC-D5 Hypercube Machine

A hypercube has 2^d identical nodes where d is the dimension of the hypercube. The 2^d nodes are labeled 0 to 2^d-1. Two nodes in a hypercube are adjacent just if their labels differ in only one bit position.

iPSC-D5 [16] is a hypercube machine with dimension five. Thirty-two microcomputers are connected so as to form a hypercube structure. Each microcomputer is referred to as a *node*. Each node consists of a single-board 286 computer with 512K bytes of RAM and 64K of ROM and has its own instructions, memory, and operating system. There are five high-speed communication channels for each node in iPSC-D5 connecting its adjacent nodes. But each node is completely independent and communicates with its neighbors by message passing, and there are no shared or global resources.

Each node connects to the "Cube Manager" which provides system management. The Cube Manager allows one to use the

iPSC as either a stand-alone concurrent computer system or as a powerful computational server within a distributed processing environment. In the latter case, the iPSC communicates with the Cube Manager through an Ethernet channel. An RS-422 diagnostic channel is also provided. Figure 4.1 shows the overall structure.

4.2 Parallel Influence Calculation

4.2.1 Sequential Influence Calculation

When using sequential computing, the influences of a game board are calculated by Procedure 4.1. "influencemap" is an 11-by-11 array storing the influences generated by a single stone as shown in Figure 1.8. "mapindexX" and "mapindexY" are indices for "influencemap". "boardindexX" and "boardindexY" are indices for the array "influence" which accumulates the influences.

The procedure scans all the board's intersections searching for stones. When a stone is found, the influences generated by the stone adds to the influences of the surrounding area. For each stone, the algorithm performs a maximum of 121 additions and subtractions. A black stone generates positive influences and a white stone generates

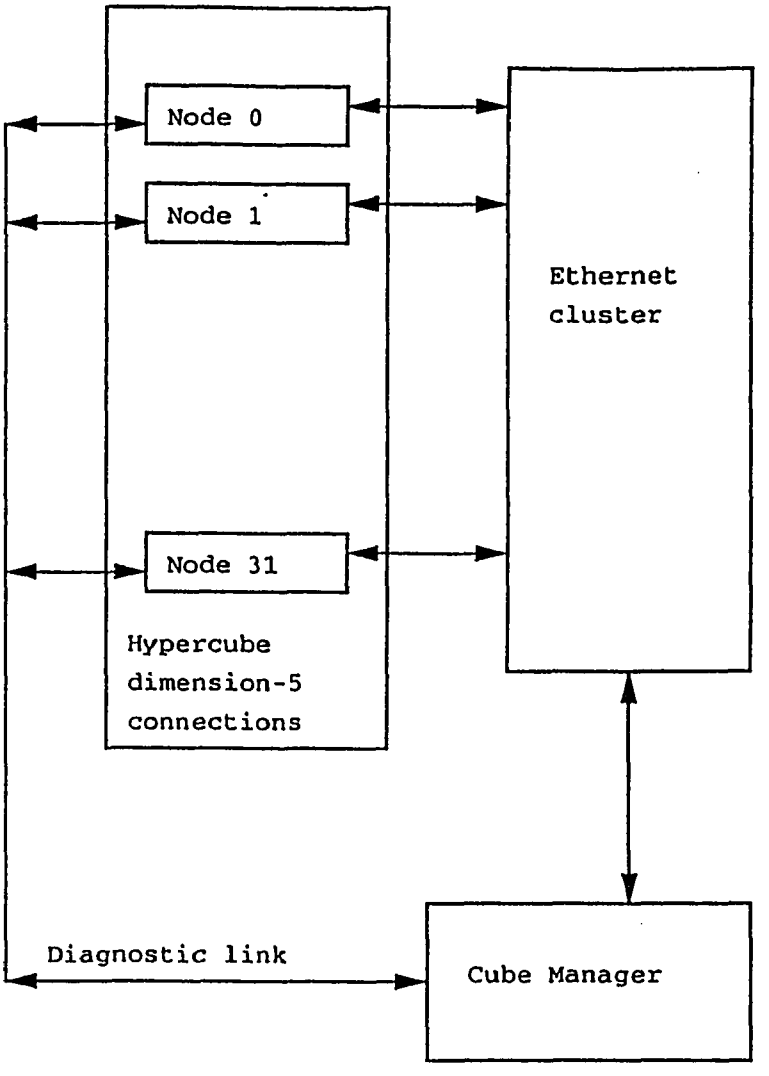


Figure 4.1 The communications between the nodes and the Cube Manager.

```
For each intersection (i,j) on the game board

 if (there is a stone at (i,j) )
  {
   for ( boardindexX=i-5, mapindexX=0; BoardindexX<=i+5;
                                  boardindexX++, mapindexX++)

     for ( boardindexY=j-5, mapindexY=0; BoardindexY<=j+5;
                                    boardindexY++, mapindexY++)

       if ( 0<=boardindexX<=20 and 0<=boardindexY<=20 )

          if (stone's color is BLACK)

               influence[boardindexX][boardindexY]+=
               influencemap[mapindexX][mapindexY];

            else influence[boardindexX][boardindexY]-=
                      influencemap[mapindexX][mapindexY];

 }
```

Procedure 4.1 Sequential influence calculation.

negative influences. n stones require up to 121n operations.

In developing our parallel computing method for influence calculation using iPSC-D5, two approaches are taken:

1. Dividing the stones into groups:
   The stones on the game board are divided into 32 groups. One processor calculates the influences generated by a given group of stones.

2. Dividing the intersections on the game board into groups:
   The 19-by-19 intersections are divided into 32 groups. One processor takes care of several intersections.

4.2.2 Dividing the Stones into Groups

In the first approach, the stones on the game board are divided into 32 groups before the parallel computing starts. Each group of stones is then sent to one processor. The processor then calculates the influences for the stones. Each processor generates a partial result. Cube additions are then performed to find the overall result. The cube additions are shown in Figure 4.2. Each cube addition merges the results from two nodes. Hence, thirty-two processors require five cube additions. Assuming that an addition takes one unit of time, the time to find the influences of n stones is:

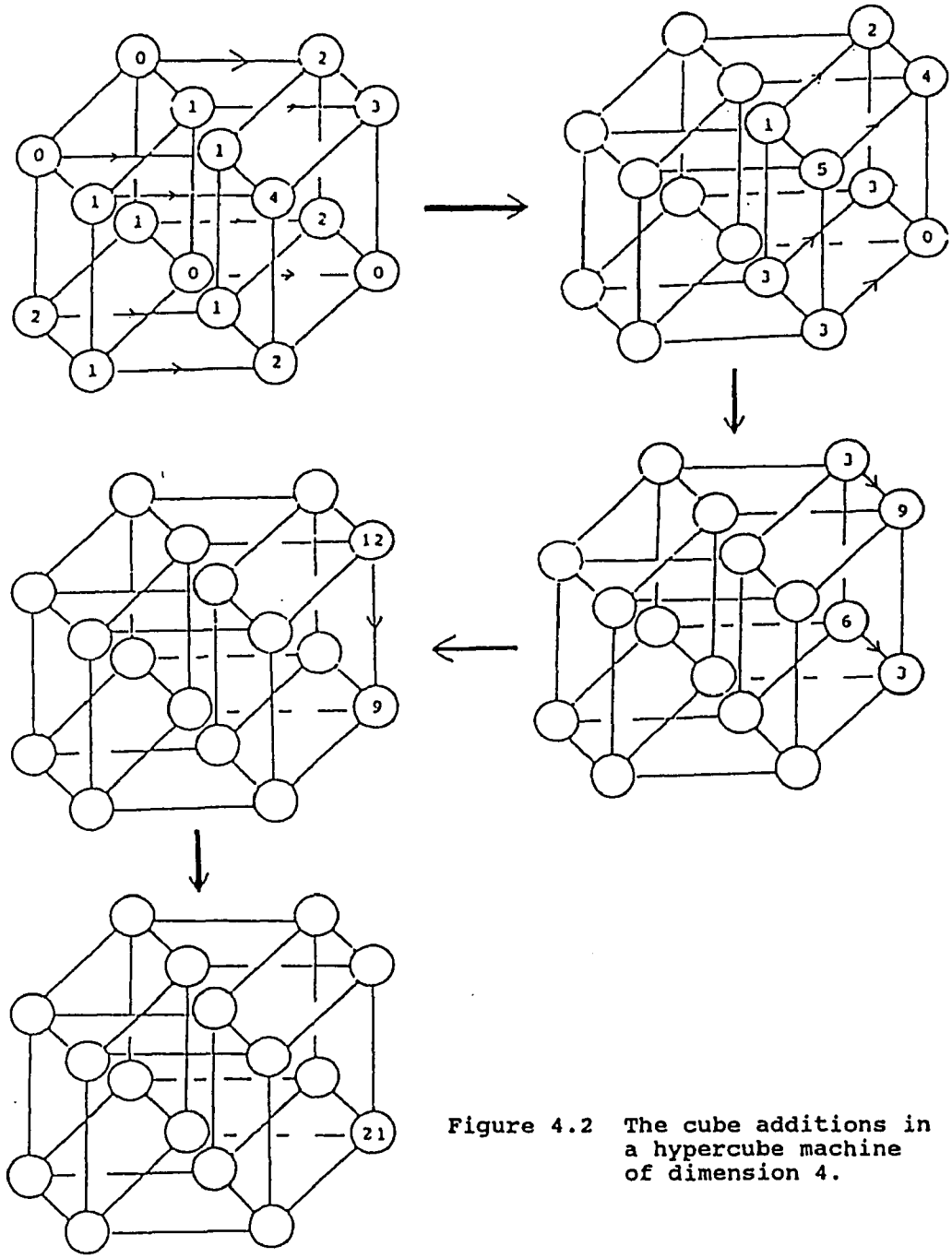


Figure 4.2 The cube additions in a hypercube machine of dimension 4.

$$11*11*\lceil n/32 \rceil + 19*19*C,$$

where n is the number of stones on the game board, C is the number of cube additions; C=5 if n>=32 and is $\lceil \log^* n \rceil$ if n<32. C=0 if n=1.

An explanation of this formula follows. When there are n stones on the game board, each node processes a maximum of $\lceil n/32 \rceil$ stones. Each stones requires 11*11 operations. Hence, the maximum number of time units for a processor to complete generating the influences and start the cube additions is $11*11*\lceil n/32 \rceil$.

The number of processors used is min(32,n), where n is the number of stones. Merging the influences of the n processors requires $\lceil \log^* n \rceil$ cube additions. Since each node provides a entire board of data, each cube addition requires 19*19 additions. 19*19*C is the number of time units needed to process the cube additions for n stones. For example, if n is 16, 11*11 additions and four cube additions are performed. Thus, processing 16 stones in parallel takes 1565 units of time, whereas sequential computing requires 11*11*16=1936 units of time. When n=1, however, the cube addition can be avoided. Table 4.1 shows the speed-up for different n.

Speed-up is calculated by dividing the time spent in

n	Speed-up	n	Speed-up
1	1.000000	100	5.286151
10	0.773163	110	5.814766
20	1.256490	120	6.343381
30	1.884735	130	6.526971
40	2.364436	140	7.029046
50	2.955545	150	7.531120
60	3.546654	160	8.033195
70	3.906827	170	8.127222
80	4.464945	180	8.605294
90	5.023063	190	9.083366

Table 4.1 Speed-up due to parallelism for different n (number of stones).

serial computing by the time spent in parallel computing excluding the communication load. When the number of stones is less than 30, the speed-up factor is less than 2. When the number of stones increases to around 200, the speed-up factor is approximately 10. During the opening game, the advantage gained from parallel computing is small because the number of stones on the game board is small. The cube additions are the dominating factor of the influence calculation and greatly reduce the speed-up factor.

4.2.3 Dividing the Intersections into Groups

With the second approach, the intersections on the game board are divided into 32 groups. Since there are 19*19 intersections, each processor handles approximately 12 ($\lceil 19*19/32 \rceil$) positions. One processor is assigned to several intersections and accumulates the influences in those locations. During the computation, the coordinates and colors of the stones are broadcast to all of the processors. The coordinates of a stone (including the virtual stone) range from 0 to 20. Since storing the two coordinates and the color of a stone requires just two bytes and one message can be declared to have up to 2*19*19 bytes, only one message is needed to broadcast the locations and colors of all the stones on the game board.

Calculating the influences

If a processor is assigned to handle position (i,j) and the stone received from the broadcast is at (x,y), the distances between the intersections (i,j) and (x,y) in both the x and y directions are calculated. The influences generated by a stone on an intersection depend on the distance between the stone and the intersection only. No influence is generated at (x,y) by the stone at (i,j) if the distance between (i,j) and (x,y) in either the x direction or the y direction is larger than five.

For the cases in which the distance in the x and y directions are both less than 5, a matrix is constructed to map the distance into the influences, as shown in Figure 4.3. Every node in the hypercube machine has such a matrix. The processor determines the influences of a location from the matrix using the distances in the x and y directions as the indices.

In Figure 4.4, we provide an example to show the sequence of the operations needed to calculate the influences. Assume that Processor 2 handles positions (1,1) and (2,3) as shown in Figure 4.4(a). Processor 3 takes care of positions (2,1) and (3,3) as shown in Figure 4.4(b). Two black stones at (1,3) and (2,2) are broadcast to all processors. The coordinates of

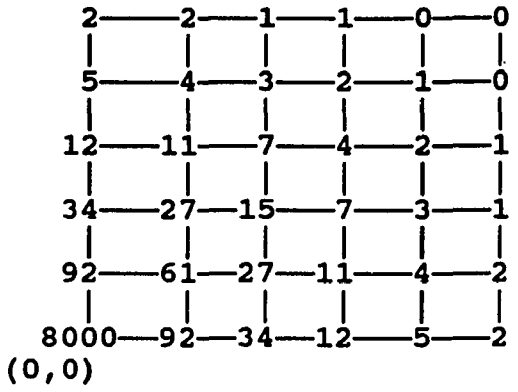
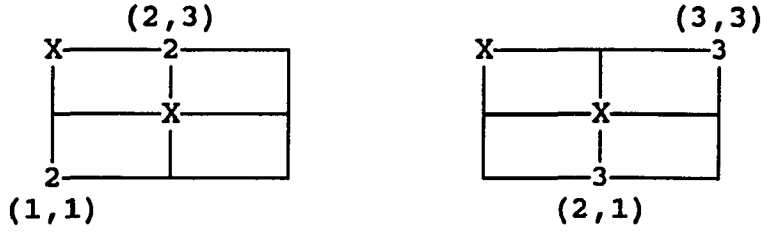


Figure 4.3 This matrix maps distances to influences.



4.4(a) 4.4(b)

Figure 4.4 An example showing the sequence of operations needed to calculate influences. Processor 2 handles (1,1) and (2,3). Processor 3 handles (2,1) and (3,3). Two black stones are on (1,3) and (2,2).

these two locations and the colors of the stones are packed into one message. The processors unpack the message to find out the locations. When Processor 2 sees (1,3) in the message, it calculates the distance from (1,3) to (2,3) as (1,0). Using 1 and 0 as the indices, the total influence, 92, is taken from the matrix shown in Figure 4.3. This influence is assigned to the location (2,3).

The same process is applied to assign an influence of 34 to the location (1,1). After processing all the locations that Processor 2 is responsible for, the stone at (2,2) is processed. This stone generates the influences 92 and 61 at the locations (2,3) and (1,1). Processor 2 adds 92 and 61 to the influences for locations (2,3) and (1,1), respectively. At the end of the influence calculation, Processor 2 sends a message containing (2,3,184) and (1,1,95) to the procedure waiting for the result. In the same manner, Processor 3 reports the influences at locations (2,1) and (3,3) as 119 and 95, respectively.

The advantage of this approach is that the influences in one location are calculated by a single processor and that, therefore, cube addition is avoided.

The assignment of the processors to the intersections

The distribution of the 32 processors to the 19*19 locations should be made as evenly as possible so that the processors all do an equal amount of work.

Calculating the optimal processing time

A stone at the central area generates the influences covering an 11-by-11 area. The influences generated by a stone close to the border cover only part of the 11-by-11 area. The intersections in the corners and the edges are processed quickly because they have fewer positions to process than those in the central area.

The influences of the stone at the intersection (0,0) cover 36 locations, and 28 processors are assigned to a single location, while 4 processors will cover 2 locations. Processing the stone at (0,0) takes two units of time, because the 28 processors with one location idle while the 4 processors finish their second location.

If a stone generates the influences covering n intersections, the n intersections are distributed evenly to the 32 processors until the intersections left behind are less then 32. Those intersections then need one more unit of time to complete processing. The optimal processing time for the n intersections is, therefore, $\lceil n/32 \rceil$. The influence area

generated by the stone at the central area takes ⌈11*11/32⌉ units of time to process optimally.

The optimal influence processing time of each intersection is accumulated and divided by 19*19. The result is the average optimal influence processing time. The average optimal influence processing time for the entire game board is 3.26.

Several ways of distributing the processors to the intersections were tried. Figure 4.5, 4.6, 4.7, and 4.8 show the different methods of distributing the processors. The number on an intersection identifies the processor assigned to the intersection. In the assignment shown in Figure 4.5 (Assignment 1), the processor for a location is selected randomly from among 32 processors.

In the assignment shown in Figure 4.6 (Assignment 2), 32 processors are arranged so as to form a four-by-eight rectangle. The rectangle then is repeatedly tiled to the 19*19 board positions. Figure 4.6(a) shows the 4-by-8 rectangle, while Figure 4.6(b) shows the assignment.

In the assignment shown in Figure 4.7 (Assignment 3), 32 processors are arranged into a six-by-six square. The positions that are left empty in the square are filled

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
19	9	3	30	4	1	12	22	14	18	16	9	17	17	27	9	11	19	6	27
18	28	7	12	30	25	4	30	13	28	6	23	7	30	19	18	13	8	3	27
17	11	6	31	3	26	29	9	24	5	31	29	20	11	28	22	21	5	27	19
16	13	9	10	28	27	14	18	0	26	16	14	0	24	22	29	18	9	18	22
15	30	29	9	28	13	15	20	18	4	0	6	16	11	4	16	23	18	27	1
14	2	22	2	17	29	1	11	31	26	16	10	25	2	25	18	29	9	12	0
13	30	5	25	21	9	0	18	10	9	12	19	31	7	22	31	20	22	30	13
12	28	14	6	3	15	23	9	1	31	9	15	1	13	17	30	29	28	22	18
11	18	6	29	16	4	15	10	23	2	23	8	25	12	19	11	9	18	30	19
10	20	30	12	20	21	20	31	20	4	25	30	21	15	18	2	20	12	15	19
9	16	13	8	19	15	21	6	25	29	10	5	0	23	11	24	1	7	4	14
8	31	16	0	20	1	13	30	27	5	21	7	27	19	2	31	24	16	18	26
7	4	20	14	17	5	30	9	21	24	6	10	4	22	19	14	25	22	15	7
6	8	20	23	10	10	16	23	8	22	26	22	27	17	13	14	4	8	25	30
5	18	27	5	3	5	28	20	5	19	24	8	14	11	21	5	0	25	29	23
4	7	10	25	22	13	16	2	9	13	8	20	9	10	25	5	8	1	0	20
3	17	27	10	5	17	8	1	12	22	9	11	7	23	12	15	17	21	29	26
2	21	1	27	1	0	30	9	8	26	7	6	2	21	20	31	26	7	25	3
1	17	21	16	24	8	17	25	12	1	6	30	14	1	16	28	30	25	17	31

Figure 4.5 Random assignment of the 32 processors to the intersections: Assignment 1.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

4.6(a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
19	**0**	**1**	**2**	**3**	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2
18	**4**	**5**	**6**	**7**	4	5	6	7	4	5	6	7	4	5	6	7	4	5	6
17	**8**	**9**	**10**	**11**	8	9	10	11	8	9	10	11	8	9	10	11	8	9	10
16	**12**	**13**	**14**	**15**	12	13	14	15	12	13	14	15	12	13	14	15	12	13	14
15	**16**	**17**	**18**	**19**	16	17	18	19	16	17	18	19	16	17	18	19	16	17	18
14	**20**	**21**	**22**	**23**	20	21	22	23	20	21	22	23	20	21	22	23	20	21	22
13	**24**	**25**	**26**	**27**	24	25	26	27	24	25	26	27	24	25	26	27	24	25	26
12	**28**	**29**	**30**	**31**	28	29	30	31	28	29	30	31	28	29	30	31	28	29	30
11	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2
10	4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7	4	5	6
9	8	9	10	11	8	9	10	11	8	9	10	11	8	9	10	11	8	9	10
8	12	13	14	15	12	13	14	15	12	13	14	15	12	13	14	15	12	13	14
7	16	17	18	19	16	17	18	19	16	17	18	19	16	17	18	19	16	17	18
6	20	21	22	23	20	21	22	23	20	21	22	23	20	21	22	23	20	21	22
5	24	25	26	27	24	25	26	27	24	25	26	27	24	25	26	27	24	25	26
4	28	29	30	31	28	29	30	31	28	29	30	31	28	29	30	31	28	29	30
3	0	1	2	3	12	13	14	15	24	25	26	27	4	5	6	7	16	17	18
2	4	5	6	7	16	17	18	19	28	29	30	31	8	9	10	11	19	20	21
1	8	9	10	11	20	21	22	23	0	1	2	3	12	13	14	15	22	23	24

4.6(b)

Figure 4.6 (a) Basic building block for Assignment 2.
(b) Assignment 2.

0	1	2	3	4	5	
6	7	8	9	10	11	
12	13	14	15	16	17	
18	19	20	21	22	23	
24	25	26	27	28	29	
30	31(	?	?	?	?)	

4.7(a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	**0**	**1**	**2**	**3**	**4**	**5**	0	1	2	3	4	5	0	1	2	3	4	5	0
2	**6**	**7**	**8**	**9**	**10**	**11**	6	7	8	9	10	11	6	7	8	9	10	11	6
3	**12**	**13**	**14**	**15**	**16**	**17**	12	13	14	15	16	17	12	13	14	15	16	17	12
4	**18**	**19**	**20**	**21**	**22**	**23**	18	19	20	21	22	23	18	19	20	21	22	23	18
5	**24**	**25**	**26**	**27**	**28**	**29**	24	25	26	27	28	29	24	25	26	27	28	29	24
6	**30**	**31**	**0**	**1**	**2**	**3**	30	31	4	5	6	7	30	31	8	9	10	11	30
7	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0
8	6	7	8	9	10	11	6	7	8	9	10	11	6	7	8	9	10	11	6
9	12	13	14	15	16	17	12	13	14	15	16	17	12	13	14	15	16	17	12
10	18	19	20	21	22	23	18	19	20	21	22	23	18	19	20	21	22	23	18
11	24	25	26	27	28	29	24	25	26	27	28	29	24	25	26	27	28	29	24
12	30	31	12	13	14	15	30	31	16	17	18	19	30	31	20	21	22	23	30
13	0	1	2	3	4	5	0	1	2	3	4	5	0	1	2	3	4	5	0
14	6	7	8	9	10	11	6	7	8	9	10	11	6	7	8	9	10	11	6
15	12	13	14	15	16	17	12	13	14	15	16	17	12	13	14	15	16	17	12
16	18	19	20	21	22	23	18	19	20	21	22	23	18	19	20	21	22	23	18
17	24	25	26	27	28	29	24	25	26	27	28	29	24	25	26	27	28	29	24
18	30	31	24	25	26	27	30	31	28	29	30	31	30	31	0	1	2	3	30
19	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

4.7(b)

Figure 4.7 (a) Basic building block for Assignment 3.
(b) Assignment 3.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
19	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7
18	11	12	13	14	15	16	17	18	19	20	21	11	12	13	14	15	16	17	18
17	23	24	25	26	27	28	29	30	31	0	22	23	24	25	26	27	28	29	30
16	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8
15	12	13	14	15	16	17	18	19	20	21	22	12	13	14	15	16	17	18	19
14	23	24	25	26	27	28	29	30	31	0	1	23	24	25	26	27	28	29	30
13	2	3	4	5	6	7	8	9	10	11	12	2	3	4	5	6	7	8	9
12	13	14	15	16	17	18	19	20	21	22	23	13	14	15	16	17	18	19	20
11	24	25	26	27	28	29	30	31	0	1	2	24	25	26	27	28	29	30	31
10	3	4	5	6	7	8	9	10	11	12	13	3	4	5	6	7	8	9	10
9	14	15	16	17	18	19	20	21	22	23	24	14	15	16	17	18	19	20	21
8	0	1	2	3	4	5	6	7	8	9	10	0	1	2	3	4	5	6	7
7	11	12	13	14	15	16	17	18	19	20	21	11	12	13	14	15	16	17	18
6	22	23	24	25	26	27	28	29	30	31	0	22	23	24	25	26	27	28	29
5	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	5	6	7	8
4	12	13	14	15	16	17	18	19	20	21	22	12	13	14	15	16	17	18	19
3	23	24	25	26	27	28	29	30	31	0	1	23	24	25	26	27	28	29	30
2	2	3	4	5	6	7	8	9	10	11	12	2	3	4	5	6	7	8	9
1	13	14	15	16	17	18	19	20	21	22	23	13	14	15	16	17	18	19	20

Figure 4.8 Assignment 4.

sequentially by the processors. The underlined area in Figure 4.7(b) shows how these empty areas are filled when processors are assigned to the board locations. The areas not contained in any square are also filled sequentially by the processors.

In the assignment shown in Figure 4.8 (Assignment 4), the processors are assigned to a square with 11*11 intersections and the square is tiled into the 19-by-19 area.

A program is created to test the uniformity of the assignments. For each position on the board, the program inspects the influenced area of the position and registers the identification of the processors which are in the area. For example, the data in Table 4.2 are generated by the program after inspecting the location (8,8) in Assignment 1.

"ID" is the identification of a processor and "NUM" is the number of intersections the processor must handle when a stone is placed at location (8,8). When a stone is placed at position (8,8), Processor 0 handles 4 positions, while Processor 18 handles 10 positions, as shown in Table 4.2. Processor 18 must do ten additions, and, therefore, it finishes later than the other processors. The other processors sit idle after they finish their respective jobs. The time to process the influences generated by a stone is determined by the processor that has the largest number of

ID	NUM	ID	NUM	ID	NUM	ID	NUM
0	4	8	2	16	5	24	3
1	3	9	9	17	3	25	5
2	3	10	4	18	10	26	2
3	0	11	6	19	4	27	1
4	3	12	3	20	5	28	3
5	3	13	3	21	2	29	5
6	3	14	2	22	5	30	4
7	3	15	2	23	5	31	6

Table 4.2 The number of intersections handled by each of the 32 processors when a stone in placed onto intersection (8,8).

positions to handle. Thus, processing the influences generated by a single stone at (8,8) takes 10 units of time.

The program also calculates the maximum amount of time needed to process the influences of a stone for every intersection. For location (8,8), this is 10 time units. The program accumulates these maximum time amounts for the entire 19*19 positions and finds the average as shown in Table 4.3.

Rows 1, 2, 3, and 4 show the result of the processor assignment in Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8, respectively. In row 1, there are fifteen intersections that require four units of time to process a stone. One hundred and five positions need six units of time and two intersections need ten units of time. The last column shows the average processing time for a single stone. This is calculated by first multiplying the number of intersections with the number of time units required to process the intersections. The program then divides the product by the number of intersections to compute the average processing time.

In Assignment 1, which is chosen randomly, the worst result is obtained, while Assignment 4 yields the best result. In the latter assignment, no intersection takes more than four units of time and the average processing time is close to the

Processor assignment	Time units (t)											Average number of time units (Σ(n*t)/n)
	0	1	2	3	4	5	6	7	8	9	10	
	Number of intersections (n)											
1	0	0	0	1	15	41	105	76	92	29	2	6.78
2	0	0	18	75	98	40	104	26	0	0	0	4.60
3	0	0	9	78	24	230	20	0	0	0	0	4.48
4	0	0	16	106	239	0	0	0	0	0	0	3.61

Table 4.3 The number of intersections and the average number of time units necessary for processing one of the intersections in various processor assignments.

optimal processing time of 3.26. Several other processor assignments were tried. They were all worse than the results shown in row 4.

The assignment in the 11-by-11 sub-matrix highlighted in Figure 4.8 is an optimal assignment. This is achieved by assigning the processors sequentially to fill all the locations in the 11-by-11 area. The processing time is $\lceil 11*11/32 \rceil$. Thus, the influence area of the stone at (6,14) is fully tiled, and, therefore, is processed optimally.

The influence area of the stone at (7,14) covers the intersections one column to the right of what (6,14) covers. The influence area of (7,14) covers the intersections from (12,19) to (12,9), which are not covered by the influence area of (6,14). The influence area of (6,14) covers the intersections from (1,19) to (1,9) that are not covered by the influence area of (7,14). The intersections from (12,19) to (12,9) are assigned to the same processors as with (1,19) to (1,9). The influence area of (7,14) is processed by the same group of processors as the influence area of (6,14). Therefore, a stone at (7,14) is also processed optimally.

In the same manner, it can be proven that all stones in the square defined by (6,14) in the upper left-hand corner and (14,6) in the lower right-hand corner (see Figure 4.9) are

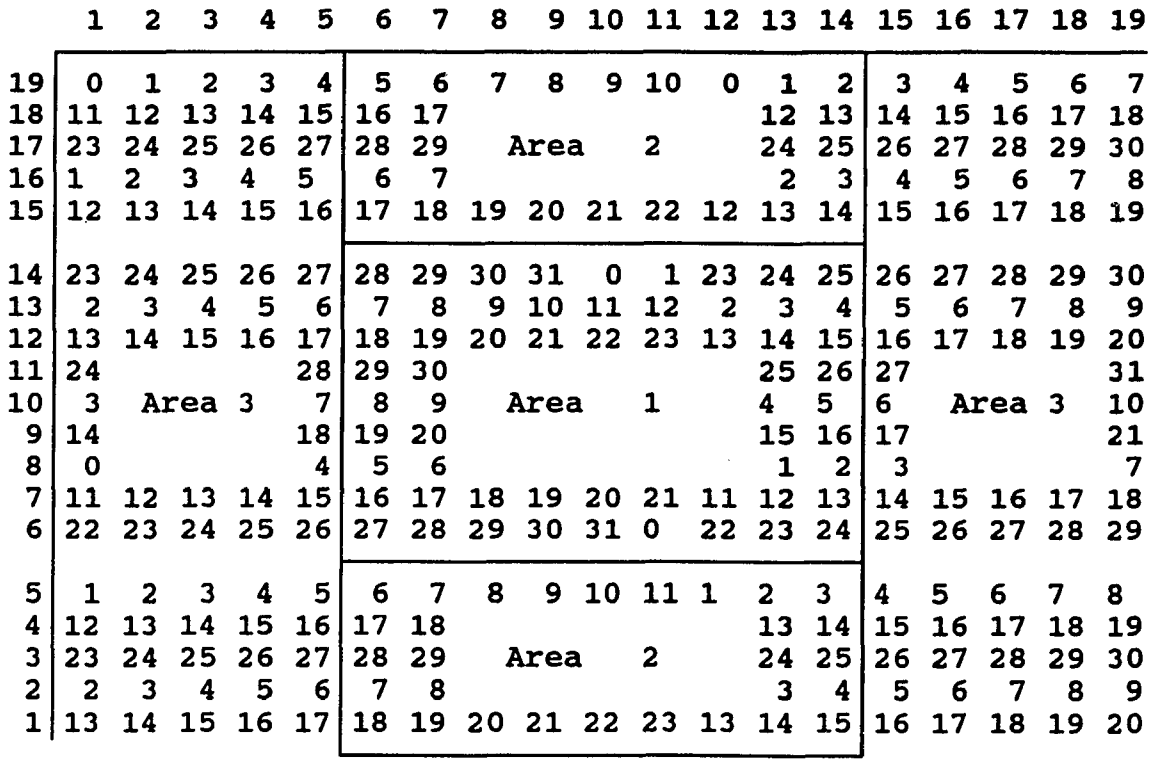


Figure 4.9 Three different areas in Assignment 4.

processed optimally. Area 1 shows this square with its 81 intersections. The stones in Area 2 are similarly optimally processed. The non-optimal cases are in Area 3. There are many methods of assigning the processors which have the same result as Assignment 4. As long as the size of the tiling is 11*11 and the assignment in the 11*11 sub-matrix is optimal, the stones in Area 1 and Area 2 are processed optimally.

When a program calculates the influences of a game board, one processor does not wait for another processor when it has finished processing a stone. The processor continues to process the stones and returns the influences of the intersections to a procedure that is waiting for the result. The total waiting time for a processor is then greatly reduced.

In a real game, the probability of putting a stone in Area 1 or Area 2 is less than that for Area 3 (the non-optimal area); a stone is more likely to be at a corner than in the central area. A simulation was done to see the performance of the different assignments in a real game. An opening game, middle game, and end game are used to find their influences. Figure 4.10, Figure 4.11, and Figure 4.12 show the games.

The results of the simulation are shown in Table 4.4. The numbers in the column "Max" give the time for the latest

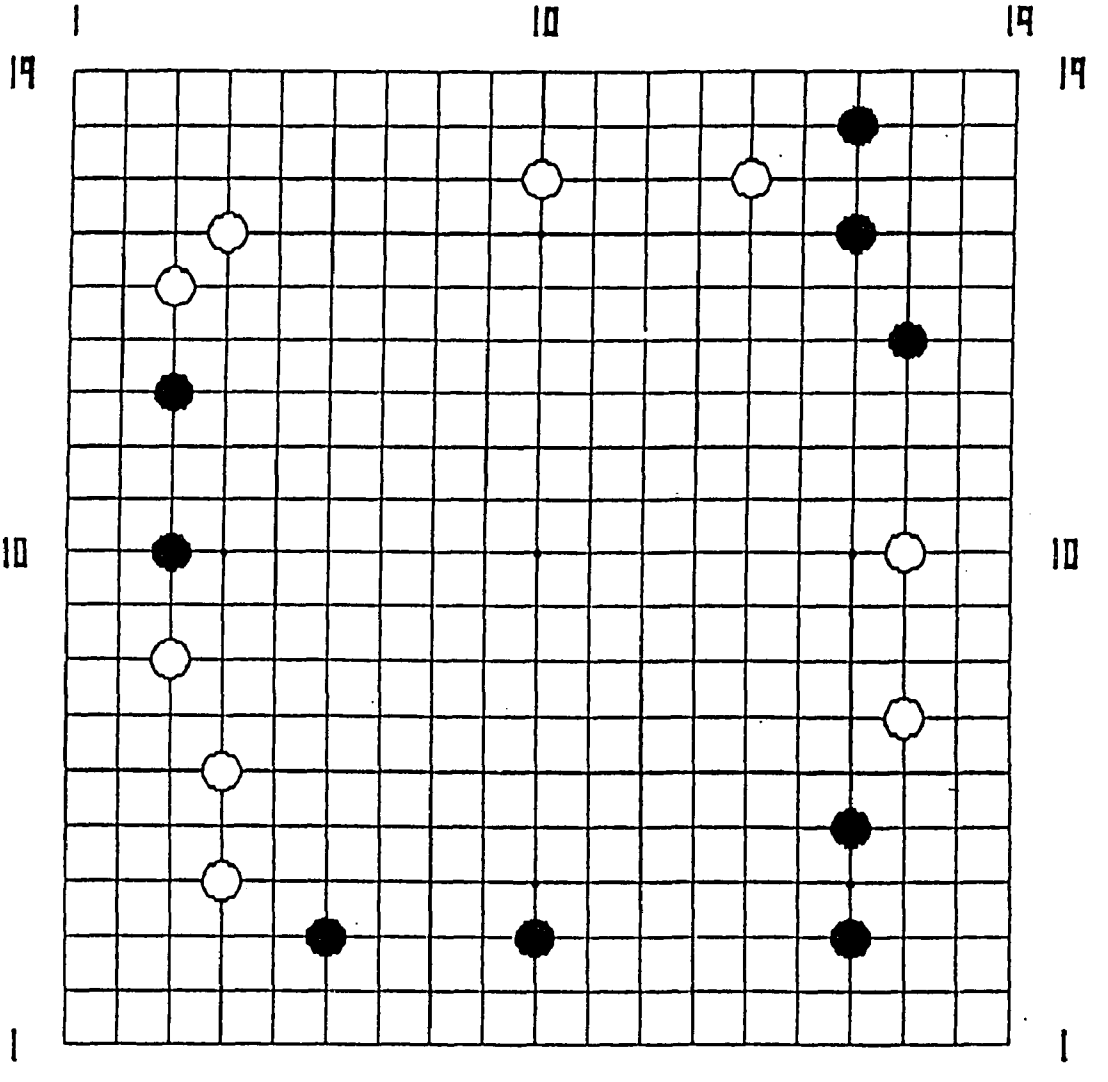


Figure 4.10 The opening game for the simulations.

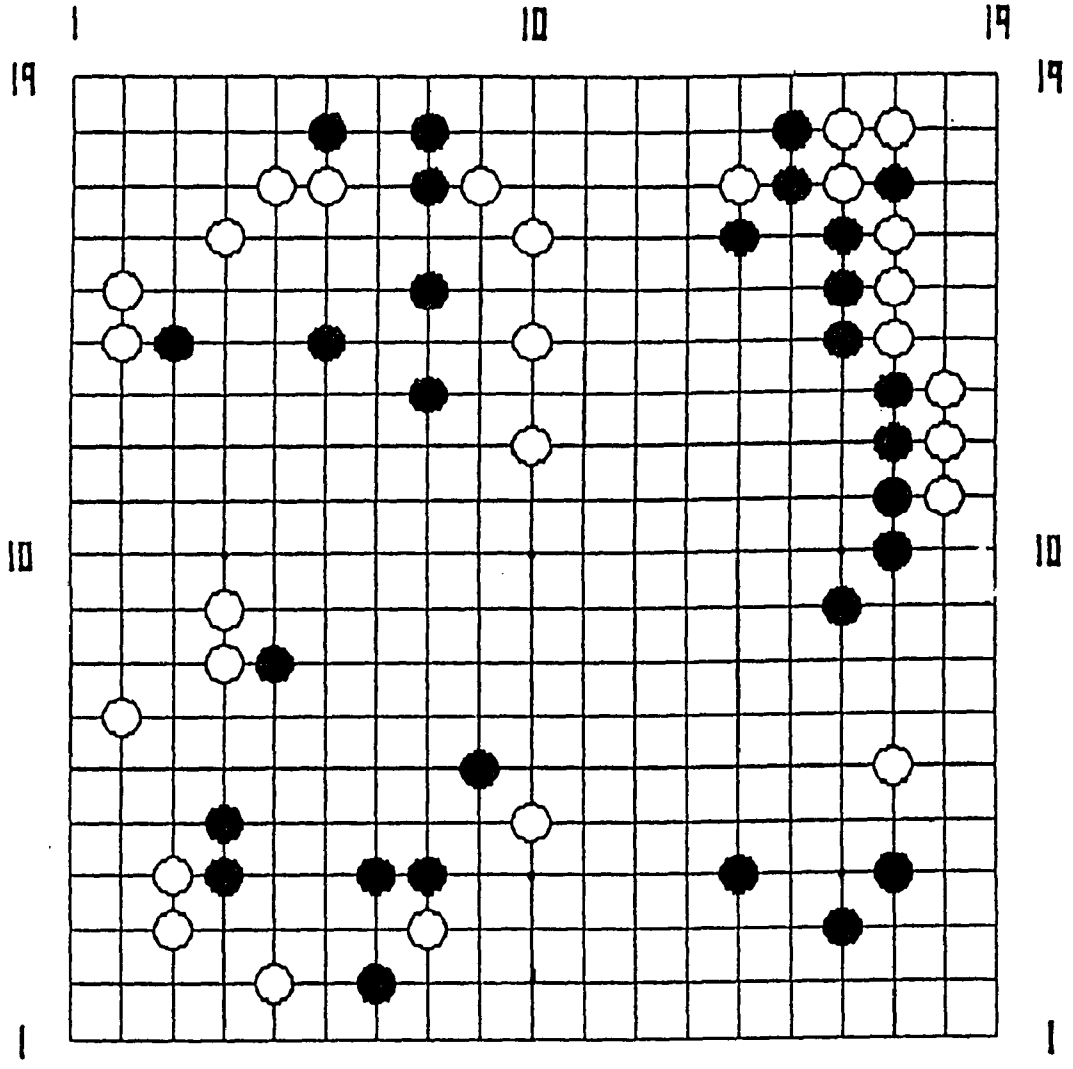


Figure 4.11 The middle game for the simulations.

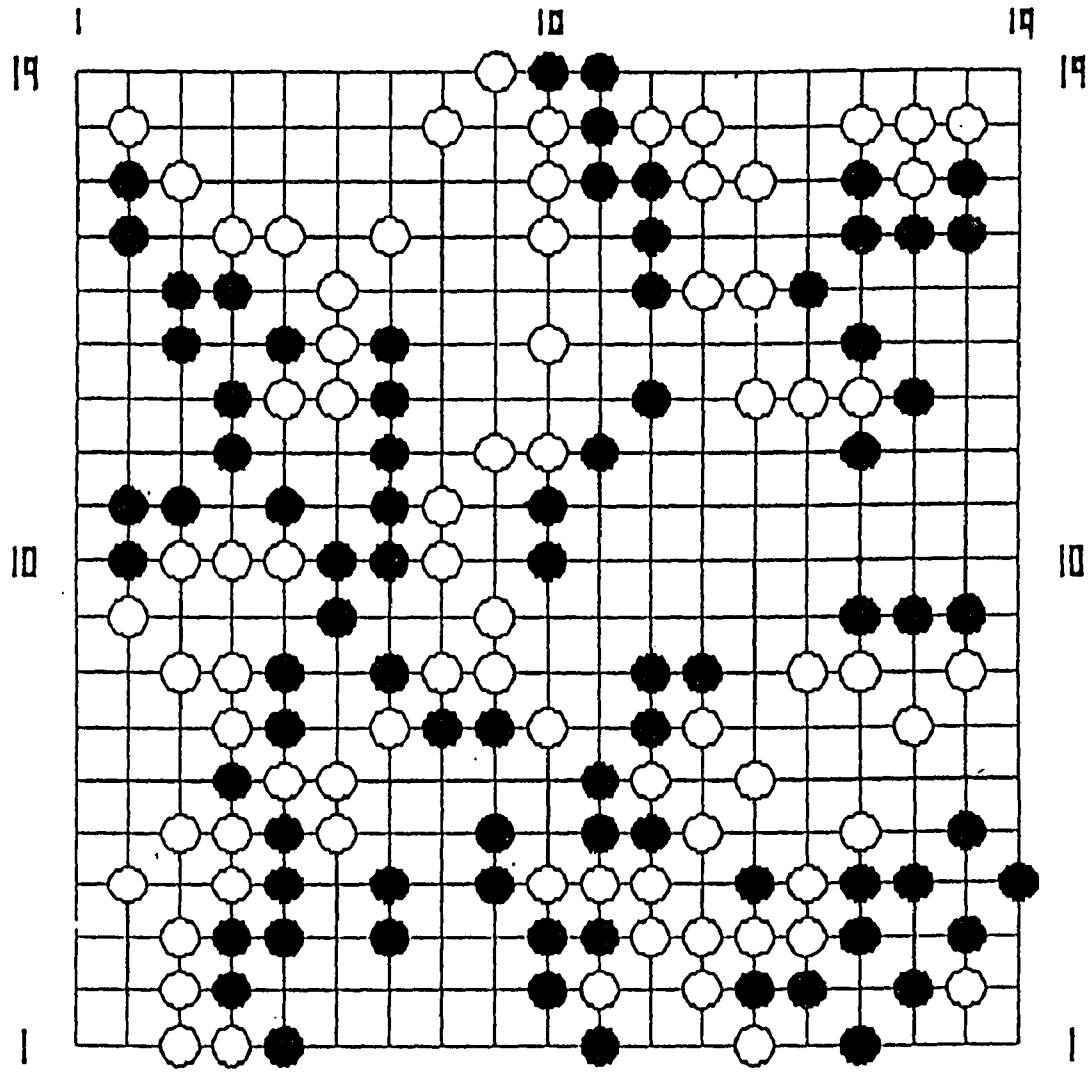


Figure 4.12 The end game for the simulations.

Opening Game

Processor assignment	Max	Min	Speed-up
1	73	22	16.23
2	55	40	**27.76**
3	66	33	23.14
4	63	21	24.24

Middle Game

Processor assignment	Max	Min	Speed-up
1	270	71	18.78
2	179	128	**28.31**
3	213	127	23.79
4	201	82	25.21

End Game

Processor assignment	Max	Min	Speed-up
1	730	165	19.63
2	518	349	**27.67**
3	527	385	27.19
4	537	278	26.68

Table 4.4 The results of the simulations in the opening game, middle game, and end game.

processor to finish processing the influences of the games. The numbers in the "Min" column show the time for the first processor to finish processing the influences of the games. Speed-up is also calculated so as to compare with the sequential machine. Speed-up was calculated after the manner of Table 4.1.

Speed-up = (Number of additions in sequential machine)/Max

Assignment 2 gives the best results in opening, middle, and end games. This processor assignment is more efficient in the corners than that of Assignment 4. Although Assignment 4 is more efficient in one-stone play (on average), its optimal area is not in the corner area. When there are more stones in the optimal area as in Figure 4.9, then Assignment 4 shows a better result. In the end game, the speed-up difference between the two assignments is smaller than that of the opening game.

4.3 A Discussion of the Parallel Computations in the Hypercube Machine

The nodes defining the same joseki pattern are currently distributed in different paths. They can be merged. Figure 4.13 shows the tree before and after merging the nodes. Node

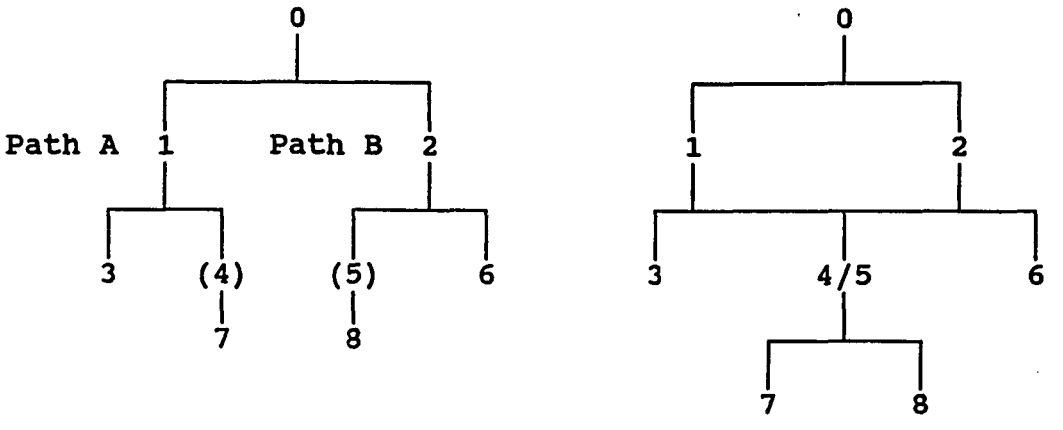


Figure 4.13 Merging two duplicated joseki nodes into one node.

4 and node 5 in Figure 4.13(a) define the same joseki pattern. In Figure 4.13(b), nodes 4/5, node 7, and node 8 will be allocated to one of the subtrees when we separate the joseki tree into different subtrees. If Path A and Path B belong to two different subtrees and node 4 and node 5 are the matching points, the matchings and the evaluations of node 7 and node 8 are performed by two processors. node 7 and node 8 in Figure 4.13(b) are processed by a single processor. Merging the nodes defining the same joseki pattern is better with sequential processing than with parallel processing.

When there are many trees to be processed, the parallel algorithm can help reduce processing time. Each subtree can be preloaded into a processor which handles a subtree independently upon receiving a game pattern from the control procedure in Cube Manager. Each processor then searches the matching points, evaluates the josekis in its subtree, and reports the best result to the control procedure. This procedure then selects the best joseki from among all the running processors. If only several subtrees generate the matching points, only the processors containing these subtrees are busy. But, the speed is no less than the time it takes to process a subtree in one processor.

During the first few moves, a parallel machine can significantly improve the performance of the joseki-finding

process, because there are a lot of subtrees to be processed. When more stones are on the game board, the number of subtrees to be processed is reduced. In this case, processors are increasingly idle. The advantage of using one subtree in one processor becomes less significant.

More research must be done to run a control procedure inside Cube Manager to search for the matching points, discover the valid josekis, and then distribute the influence evaluation to the nodes using the results of Section 4.1. The control procedure finds the valid josekis and sends the stones on the game board to the 32 processors for evaluation. Although influence calculation is fast, loading a tree from disk to Cube Manager and the communication between the control procedure and the nodes in the hypercube machine are the major factors affecting efficiency. This method may not be faster than the method which spreads one subtree to one processor. To gain the advantages of both approaches, a control procedure can use one subtree in one processor at the beginning of the game. When there are only a few subtrees left to be processed, it tries to find the valid josekis from the control procedure in Cube Manager and processes the influences in parallel using 32 processors.

CHAPTER 5

Designing Hardware Circuits to Provide the Computational Power for Joseki Processing

A custom-designed circuit provides our joseki search algorithm with a tremendous amount of computational power. In this chapter, we design the logic circuits to compare stones on the game board with the nodes in the joseki tree, detect a killing condition, detect the existence of the interference stones, and compute the influences of a game board. Not only can these circuits be used in the opening game, but they are also suitable for the middle game and end game. These circuits are then simulated to see the results of their design. At the conclusion of this chapter, the design of a joseki processor is discussed.

5.1 The Dead-Link Detector

A dead link is a link without liberty and must be removed

from the game board immediately. Discovering a dead link requires identifying the stone(s) in a link and searching the liberty for the link. Because a dead link always exists adjacent to the last stone played, our software algorithm detects a dead link by first identifying the stones in the links which are adjacent to the last stone and then finding the liberties for the links. If there is no dead link, the link containing the last stone is then checked. If this link is killed, placing the last stone is a suicide move. (Since there is no suicide move in any joseki, our program does not check for suicide moves.)

Dead-link detection must be executed after every move in order to find an updated game board condition. JOSEKI_FINDER solves this problem by pre-processing and storing the results in "killing_table" to save processing time. (For details, see Section 2.3.2.) However, the array "killing_table" and some extra operations are required to process a killing node. The problem is even worse when we apply the look-ahead tree search for the middle game or end game. In the look-ahead tree search, the dead-link detection must be executed for every node in the search tree. Because the next moves which are examined in the look-ahead tree search are dynamically generated by a move generation procedure and no predefined move sequences (such as josekis for the opening game) are available, the pre-processing strategy used by our joseki

processing program cannot be used here.

5.1.1 Implementation

The *dead-link detector* - our hardware circuit to discover dead links - requires no more than five I/O accesses to detect a dead link and remove it. The detector consists of 19*19 circuit cells. One circuit cell handles an intersection. The detector is designed based on the following considerations:

(1) An empty intersection cannot be killed,
(2) A stone adjacent to an empty intersection has liberty,
(3) If one stone in a link has one liberty, this liberty should be propagated to all the stones in the link.

Figure 5.1 shows the logic circuit implementation of these three requirements. The circuit cell generates a 1 at "AliveU" only if:

(1) it is an empty intersection,
(2) it is adjacent to an empty intersection,
(3) it receives a "Neighbor_Alive"=1 from its neighbor and the stone at the neighbor has the same color.

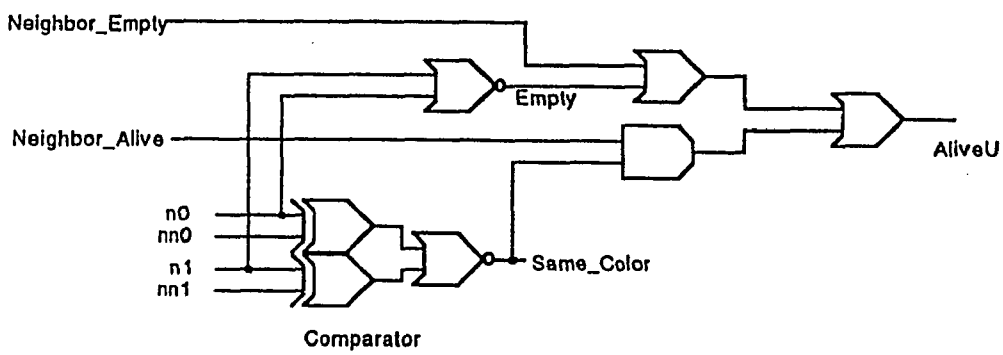


Figure 5.1 The logic circuit which generates the AliveU signal for an intersection.

If a circuit does not set "AliveU" to 1, a dead stone is in the circuit.

(n1,n0) are two bits from memory units representing the color of an intersection. In our entire circuit design, (0,1) represents BLACK, (1,0) represents WHITE, and (0,0) represents "empty". (nn1,nn0) are two bits representing a neighbor's color. "Neighbor_Alive" and "Neighbor_Empty" are the neighbor's "Alive" and "empty" signals. In the circuit, if either (n0,n1) = (0,0) or (nn0,nn1) = (0,0), "AliveU" is set to 1. Also, if n0=nn0 and n1=nn1, the comparator sets the signal "Same_Color" to 1 which enables the neighbor's alive signal "Neighbor_Alive" which is then propagated to "AliveU".

Each circuit cell consists of four such circuits for the four neighbors as shown in Figure 5.2. A 4-input OR gate collects all the alive signals ("AliveL", "AliveR", "AliveU", and "AliveB") from the four neighbors. If any of the alive signals is activated, then "AliveT" signal is set to 1.

5.1.2 Circuit Analysis

When two circuit cells, cell P and cell Q, are connected, a feedback loop is formed by the gates A, B, C, D, E, and F as shown in Figure 5.3. A feedback loop in a circuit might generate an oscillating state. Along the loop, if an output

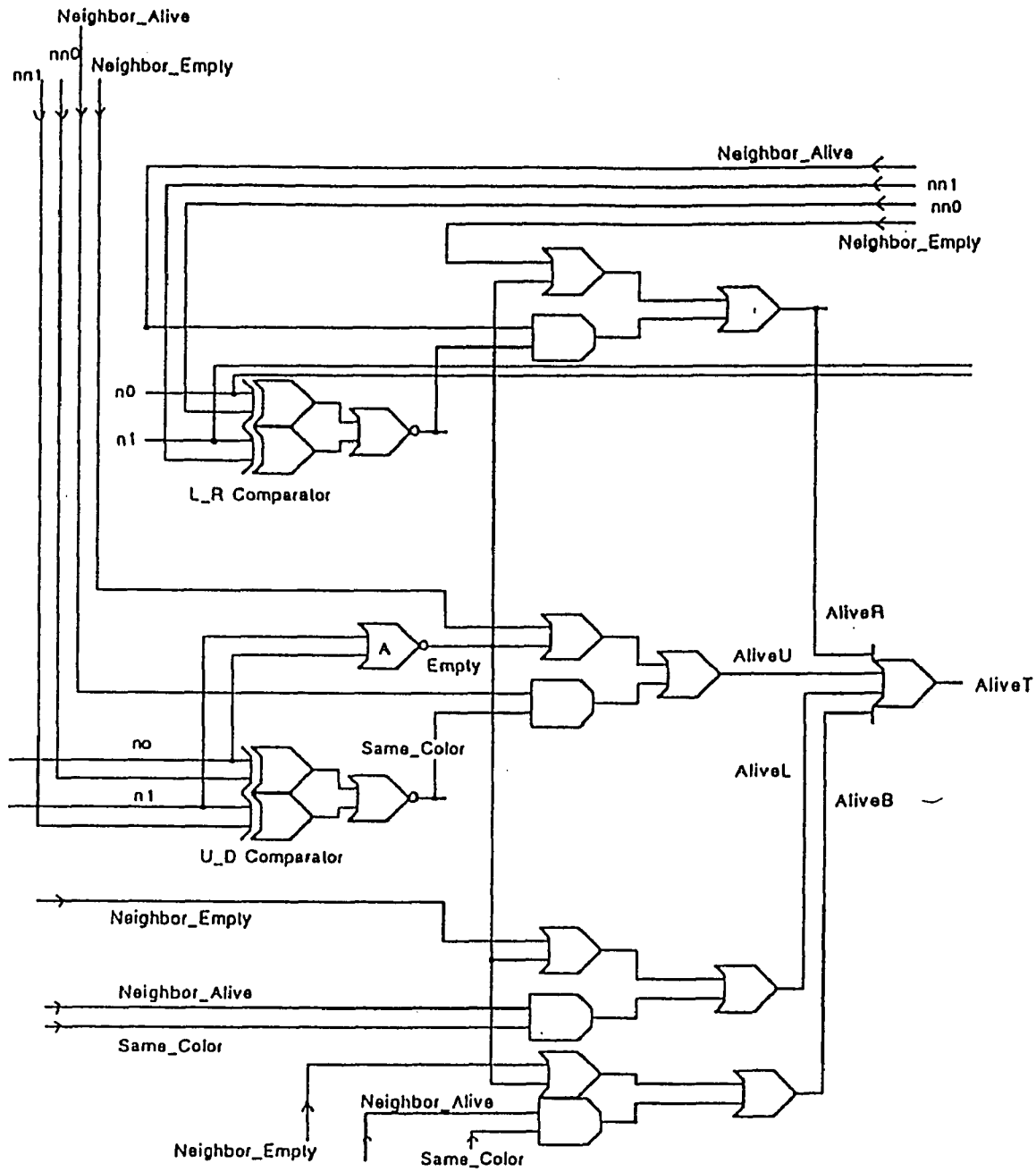


Figure 5.2 A circuit cell receives the signals from its four neighbors.

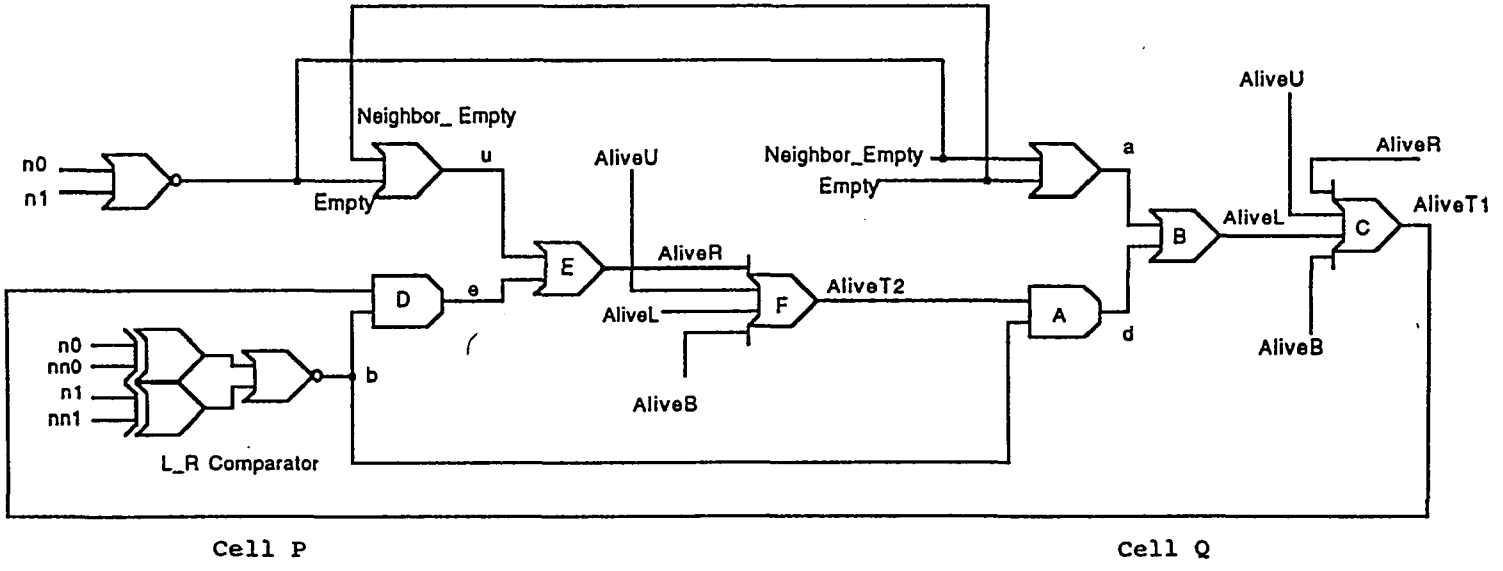


Figure 5.3 After two cells are connected, a feedback loop is formed.

from a gate generates a 1 (0) state and the 1 (0) state then generates a 0 (1) state, an oscillating output is generated through the loop.

In the feedback loop in Figure 5.3, if "AliveR", "AliveU", or "AliveB" at gates C is 1, then "AliveT1" is 1. Similarly, if "AliveU", "AliveL", or "AliveB" at gate F is 1, then "AliveT2" is 1. Also, if a=1, "AliveT1" is 1; if u=1, "AliveT2" is 1; and, if b=0, then d=0 and e=0. If the output of a gate in the loop is fixed by inputs not in the feedback loop, an oscillation through the loop cannot be generated. Assuming that "AliveR", "AliveU", and "AliveB" at gate C and "AliveU", "AliveL", and "AliveB" at gate F are all at the 0 state, "AliveT1" is always equal to "AliveL" and "AliveT2" is always equal to "AliveR". We also set a=0, u=0, and b=1, the reverse of the above assignments to these Boolean variables, so that an oscillation can be generated.

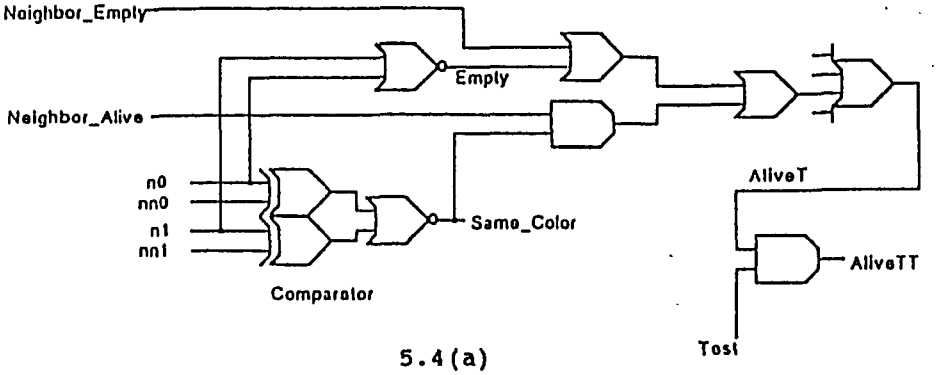
Assume that the propagation delay from "AliveT2" to "AliveT1" is dt1 and the propagation delay from "AliveT1" to "AliveT2" is dt2. Under these assumptions, the circuit has an oscillation if the initial state of either "AliveT1" or "AliveT2" is 0 and the initial state of the other is 1. For example, assume that the initial states of "AliveT1" and "AliveT2" are 1 and 0, respectively. "AliveT2" is set to 1 by "AliveT1" at time dt2 and "AliveT1" is set to 0 by "AliveT2"

at time dt1. At time dt1+dt2, "AliveT2" is set to 0 by "AliveT1" which is set to 0 at time dt1. "AliveT1" is set to 1 by "AliveT2" at the same time. The process continues. Thus, "AliveT1" goes from 0 to 1 and from 1 to 0 at time dt1+dt2: An oscillation occurs in the circuit. However, if the initial states of "AliveT1" and "AliveT2" are both 0 (1), they are stable at 0 (1).

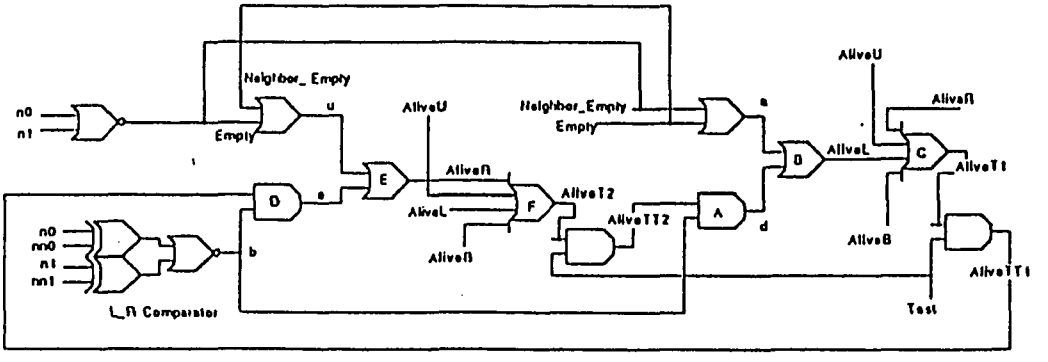
The assumptions we made in the above analysis occur when cell P and cell Q have two stones with the same color and no alive signal is propagated to them from their neighbors. That is, these two stones are enclosed by enemy stones; they form a dead link.

The initial states of "AliveT1" and "AliveT2" are 1 before a killing occurs. When the stones on the circuit cells are killed, these two circuits form a feedback loop keeping "AliveT1" and "AliveT2" as 1 in its initial state, though no alive signal is propagated to the link. The circuit at Figure 5.3 fails to generate a 0 state at "AliveT1" and "AliveT2" when their stones are enclosed by enemy stones. In fact, the circuit fails to generate a 0 for a dead link which has more than one stone.

An AND gate with output "AliveTT" is added to the cell as shown in Figure 5.4(a). Since the normal state of the "Test"



5.4(a)



5.4(b)

Figure 5.4 (a) Adding an AND gate and Test signal to set the initial states of AliveT1 and AliveT2 to 0.
(b) The initial states of AliveTT1 and AliveTT2 are set to 0 by Test signal.

signal is 0, the output of "AliveTT" will also be 0. Therefore, "AliveT1" and "AliveT2" in Figure 5.4(b) are also set to 0 through the feedback loop. When we try to detect a killing condition, "Test" is set from 0 to 1. Because the initial state of "AliveT1" and "AliveT2" is 0 now when "Test" is set to 1, the circuit cell which has its stone killed generates a 0 at "AliveT1" and "AliveT2", and, therefore, at "AliveTT1" and AliveTT2". "Test" is then reset to 0 after the killing detection.

5.1.3 Detecting a Non-Liberty Killing

There are 3 possible cases when a killing occurs:

(1) The last stone has liberty.
(2) The last stone has no liberty and it kills other stone(s).
(3) The last stone has no liberty and it does not kill other stone(s); it is a suicide move.

Figure 5.5 shows these three cases. In each case, the grayed stone is the last stone which generates the killing conditions. Figure 5.5(a) shows case (1). Figure 5.5(b) and Figure 5.5(c) show case (2) and Figure 5.5(d) shows case (3). The grayed stones in Figure 5.5(a) and Figure 5.5(b) are black stones and the grayed stones in Figure 5.5(c) and Figure

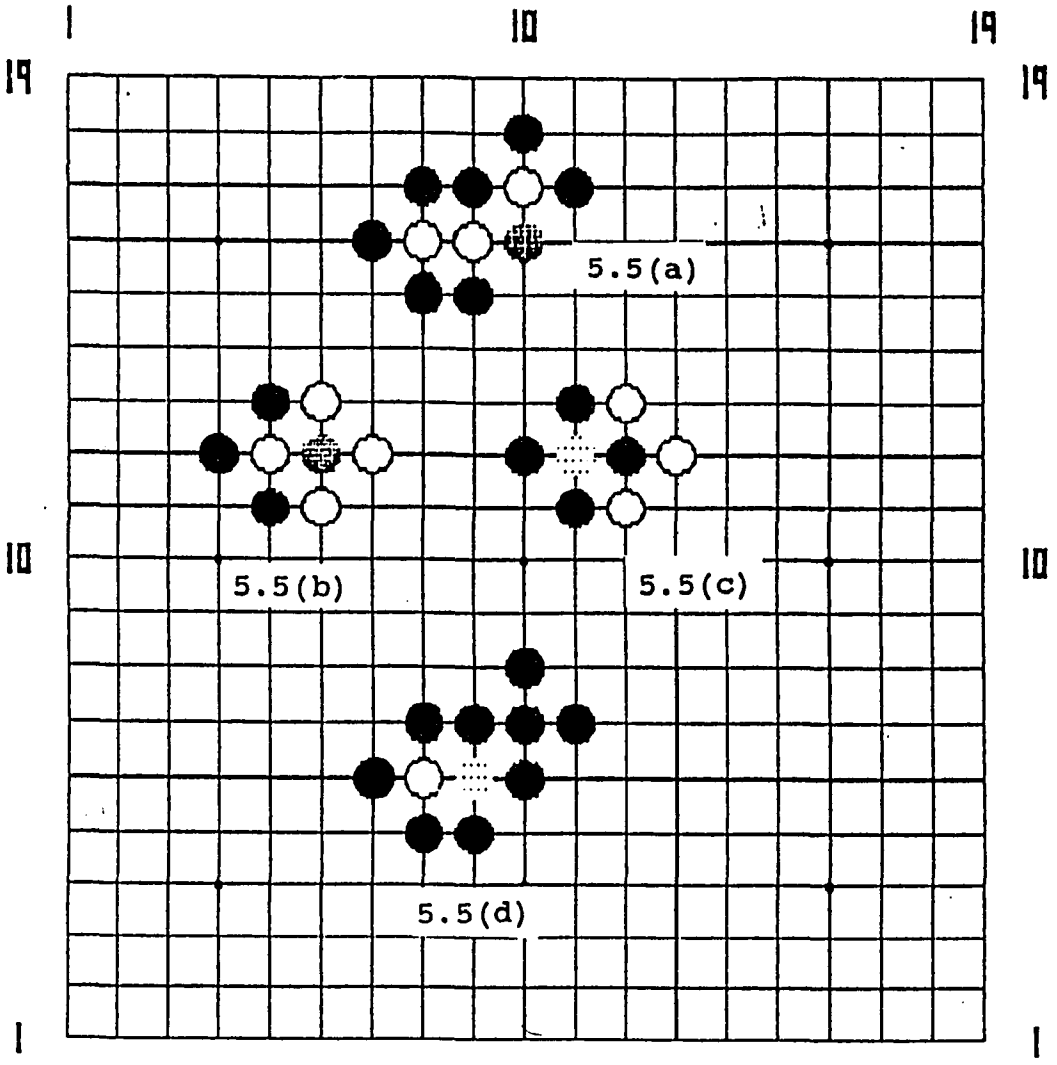


Figure 5.5 Four differnt killing configurations.

5.5(d) are white stones. The configurations of the stones in Figure 5.5(b) and Figure 5.5(c) are exactly the same. However, in Figure 5.5(b), the stone at (5,12) is killed, while in Figure 5.5(c), the stone at (12,12) is killed. A circuit cannot distinguish the killing of Figure 5.5(b) from that of Figure 5.5(c), without referring to the position of the last stone. The circuit shown in Figure 5.4 does not use any information about the last stone: It can therefore not distinguish between these two cases.

The circuit of Figure 5.4 is modified as shown in Figure 5.6 to distinguish between the two cases shown in Figure 5.5(b) and 5.5(c), respectively. When we load a stone into cell (x,y), the "Select" signal in cell (x,y) is set to 1. When the last stone is placed in a circuit cell, a "Select" signal of 1 is generated. When "STest" and "Select" are both 1, the circuit is forced to generate a 1 in "Alive". Therefore, the link containing the last stone is alive under all input conditions.

An AND gate (gate A in Figure 5.6) is used to collect the "Alive" signals of all cells. When any cell generates a 0 at "Alive", i.e., any stone on the board is killed, "Killing" is set to 1. A stone can kill two or more links; our circuit handles all dead-links correctly.

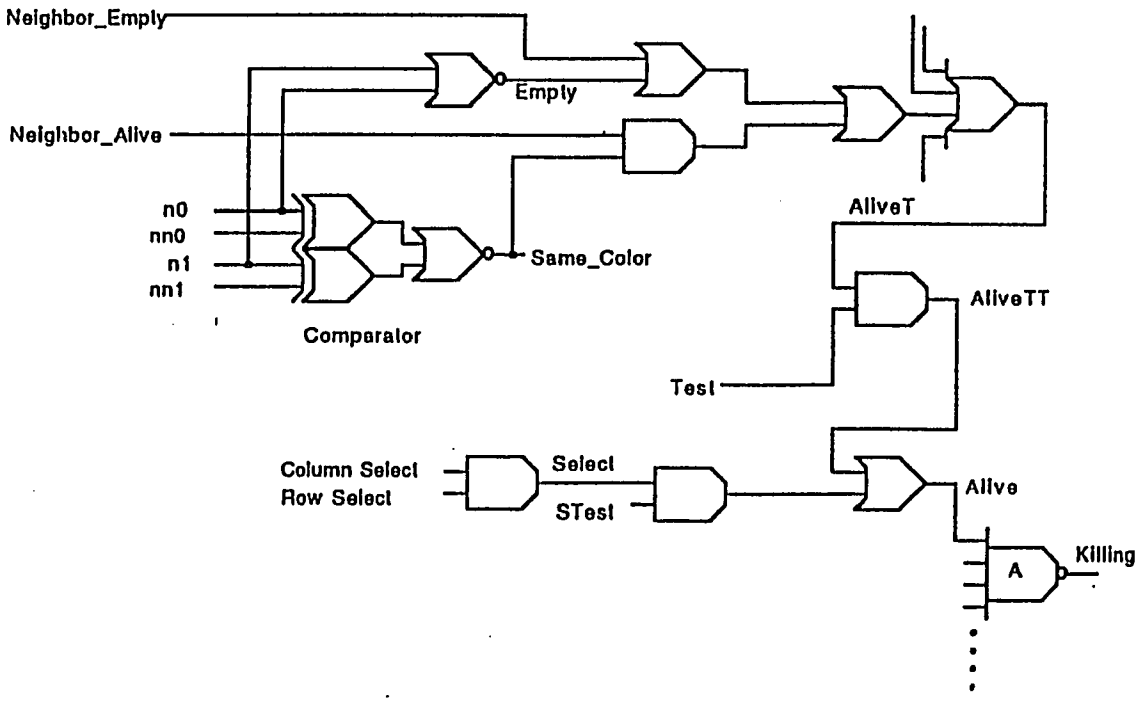


Figure 5.6 Adding an OR gate and STest signal forcing the last stone to be alive.

The circuit is used to differentiate between the 4 cases referred to earlier (Figure 5.5 and accompanying text in Section 5.1.3). After placing a stone into a circuit cell, inspect the "Killing" and the "Alive" signal of the circuit cell. If a killing happens ("Killing" is 1) and "Alive" is 0, set "STest" to 1. If "Killing" is still 1, the link which still has "Alive" as 0 is the dead link. However, if "Killing" is changed to 0, the link which connects the last stone is the only link killed before "STest" is set to a 1 and placing the last stone is a suicide move.

5.1.4 Removing the Killed Link

Every circuit cell which has its stone killed generates a 0 at "Alive". This signal is used to reset the stone in the cell to "empty" as shown in Figure 5.7. When the input "Remove" is set to 1 and "Alive" is 0, (n0,n1) is reset to (0,0). When "Killing" is 1 and "Alive" is 0, a suicide detection is required before "Remove" is set to 1. If there is no suicide move, the link containing the last stone should not be removed. Hence, "STest" is set to 1 forcing "Alive" to 1 for all stones linked to the last stone. Therefore, this link will not be removed when "Remove" is set to 1.

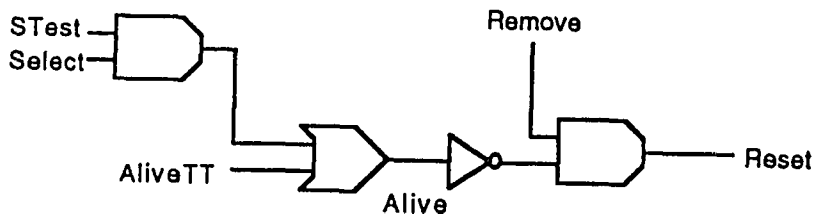


Figure 5.7 Logic circuit to remove a stone.

5.1.5 Storing and Restoring the Game Board

JOSEKI_FINDER restores a game board when the matching procedure backtracks from a child node. If the information associated with the child node indicates that no killing has occurred, the algorithm simply removes the stone given by the child node. Otherwise, in addition to removing the stone, the stones which were killed must be restored; this is accomplished by backtracking. In our software implementation, we store these killed nodes in an array and restore them by searching the array when backtracking.

We can eliminate the array and these storing and restoring operations by providing every circuit cell with a stack. Figure 5.8 shows the block diagram of the circuit. A multiplexer selects the data n0 and n1 from either the stacks or the data bus. When "Pop" is 1, n0 and n1 are popped from the stacks. When "Push" is 1, n0 and n1 are pushed onto the stacks. When a killing happens, every stone is pushed onto the stacks and is later restored by popping the stones from the stacks.

5.1.6 Simulation and Testing

Figure 5.9 and 5.10 show the circuits we built for

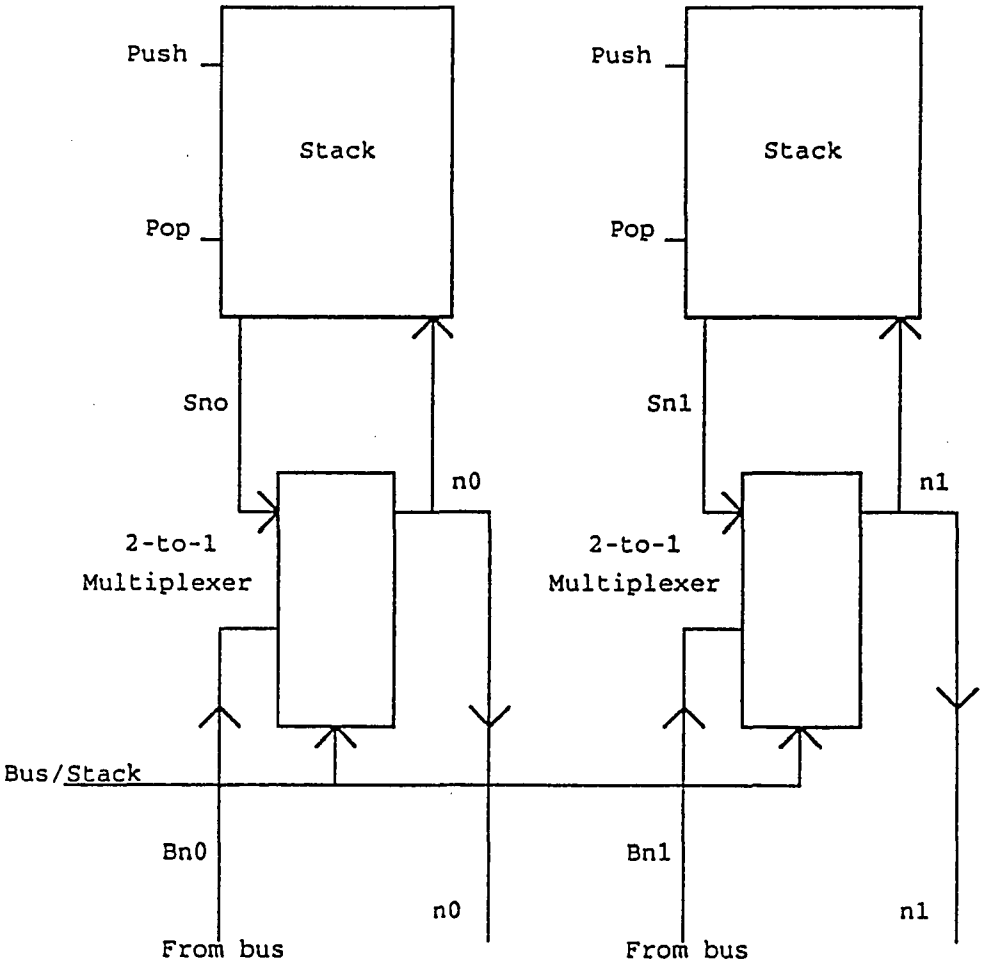


Figure 5.8 The stacks for storing and restoring a stone.

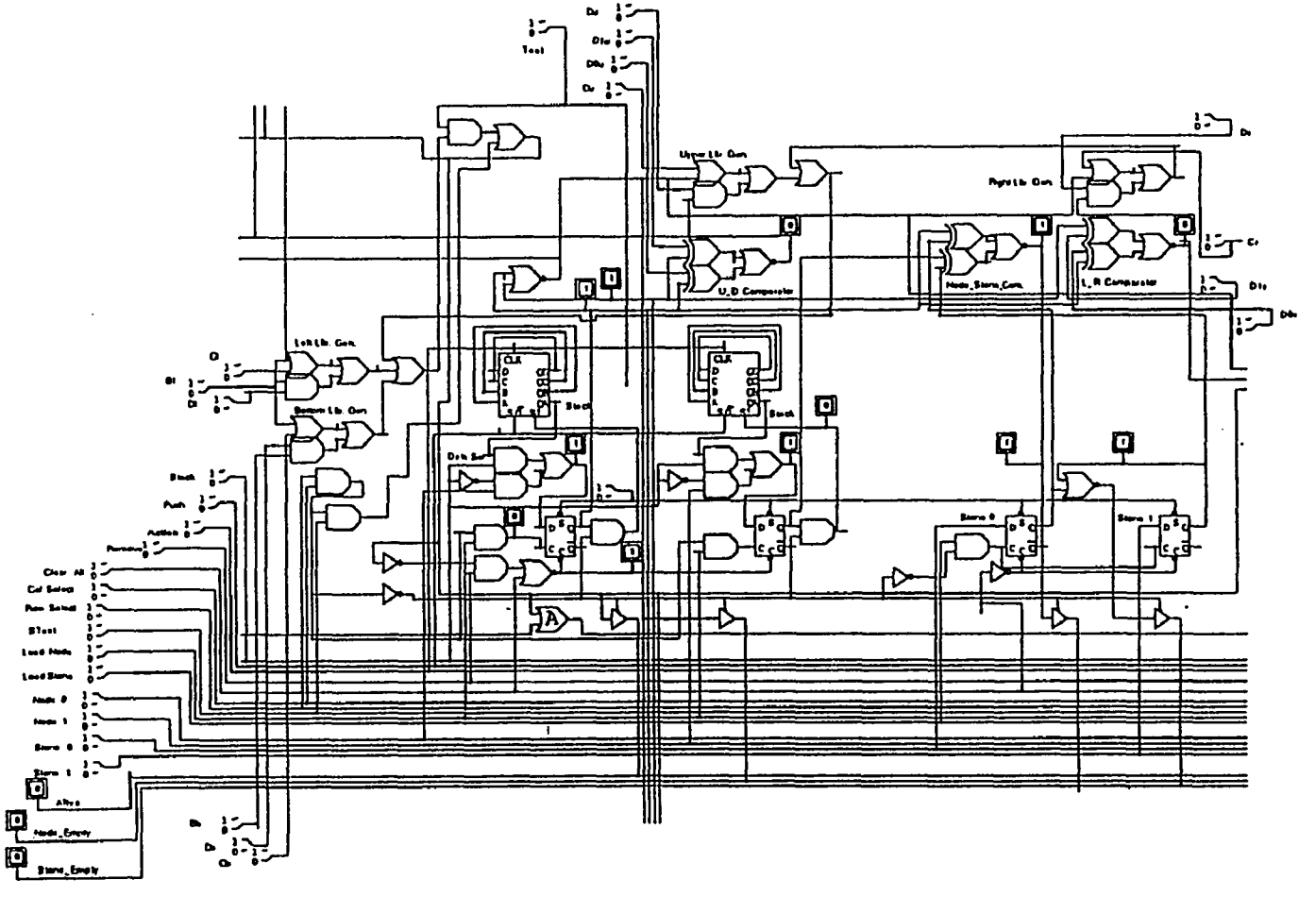


Figure 5.9 Circuit cell of the dead-link detector.

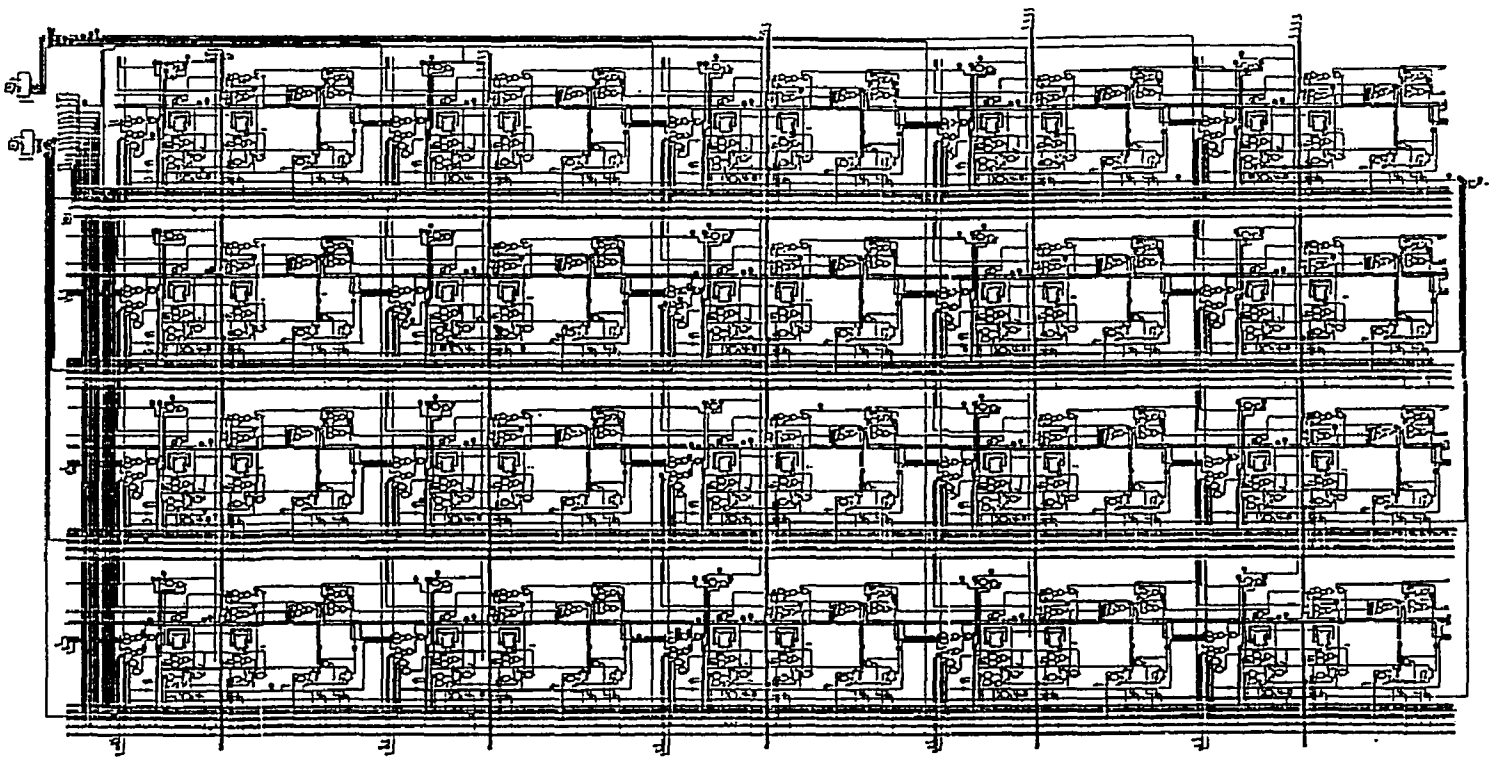


Figure 5.10 A dead-link detector for testing which consists of 4*5 cells.

simulation. Figure 5.9 shows a circuit cell while Figure 5.10 shows the circuit constructed by 4*5 circuit cells. All of the simulations in this chapter are carried out by LOGIMAC™ which provides only a few devices for simulation.

LOGIMAC™ has available D flip-flops, 4-bit registers, and left-shift registers, but neither left-right-shift registers nor stacks. We are able to simulate these last two devices as follows: The left-shift register shown in Figure 5.11(a) allows four data to be loaded in parallel from inputs A, B, C, and D, when S/L is 0. It also allows a serial input from ST when S/L is 1. Outputs QB, QC, and QD are connected to inputs A, B, and C, respectively, as shown in Figure 5.11(b). By setting S/L to 0, the device performs the shift-right operation. We now have a left-right-shift register which, in turn, can be used as a stack. All stones on the game board are pushed onto the stacks when the last stone generates a killing. Only the stones before the last stone are stored so that the configuration before the killing can be restored from the stack, i.e., the last stone should not be pushed onto the stacks even it has loaded into a now-selected cell. (This cell has its Select signal set to 1.) AND gate A in Figure 5.11(b) generates a 0 for input to the stack when the cell is selected. This stops the last stone from being pushed onto the stacks. The multiplexer which selects the input data from either the stacks or the data bus is also shown under the

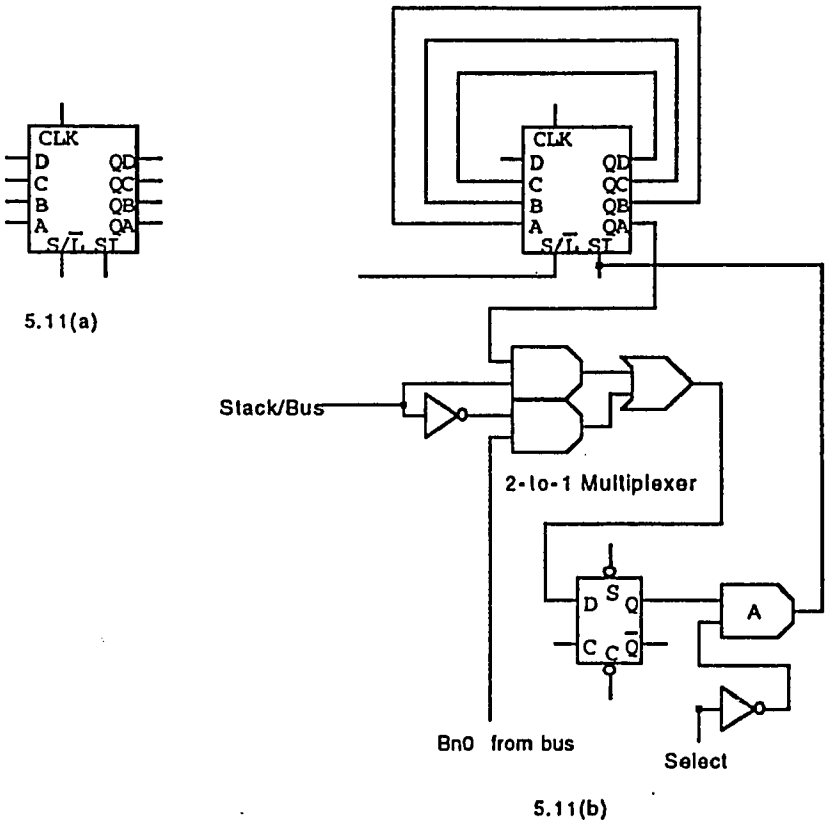


Figure 5.11 The stack circuit and its central component.

left-right-shift register.

Before the joseki search begins, some stones are ready on the game board waiting to be compared with the joseki patterns; these stones are called *S-stones*. These stones shows the game board to be processed. The stones which form the various joseki patterns and are indicated by the node in the joseki tree are called *N-stones*. If all N-stones match some S-stones on the game board, a joseki pattern is found among the S-stones.

S-stones generate a static game pattern (i.e., one which does not change during the joseki processing). A circuit cell stores an S-stone into the two D flip-flops in the right-hand side of Figure 5.9. N-stones are loaded one-by-one as indicated by the nodes in the joseki tree during the processing. Two D flip-flops in the left-hand side of Figure 5.9 are for an N-stone. "Killing" is generated by N-stones. Therefore, only N-stones have stacks. The Node_Stone Comparator shown in Figure 5.9 generates the result of matching the S-stone with the N-stone in the cell. OR gate A is used to collect the alive signal in one row of circuit cells. The circuit also generates three signals: "Alive", "Stone_Empty", and "Node_Empty".

Two 3-to-8 decoders, "X_Coordinate" and "Y_Coordinate",

in the upper left-hand side of Figure 5.10 are used to select a circuit cell. A bus connects all 5*4 circuit cells. The 4-input NAND gate in the right-hand side of Figure 5.10 collects all the "Alive" signals from each row and generates the "Killing" signal.

Different test cases are simulated by the circuit in Figure 5.10. One of the test cases is shown in Figure 5.5. The circuit correctly removes the dead links shown in Figure 5.5(b) and Figure 5.5(c), and identifies the suicide move in Figure 5.5(d).

5.2 Influence Calculation Circuit

Calculating the influences of a game board sequentially requires 11*11*n additions and subtractions, where n is the number of stones on the game board. Our influence calculation circuit requires n I/O accesses and n additions to perform the influence calculations. The circuit is designed using the method developed in Section 4.2 in which a single intersection is handled by a processor. When we developed the parallel influence processing algorithms for the iPSC-D5, only 32 processors are available; therefore, a processor has to handle several intersections. In contrast, we use 19*19 circuit cells to cover the game board. A circuit cell handles an intersection and a bus connects all the cells.

5.2.1 Implementation

Figure 5.12 shows the block diagram of a circuit cell. Registers 1 and 2 store the x and y coordinates of a circuit cell. For example, if a circuit cell processes the intersection (10, 8), register 1 is preset to 10, while register 2 is preset to 8. Block A accepts the coordinates of the stone which generates the influences, and then generates the x and y distances between the stone and the cell. Block C is preset to contain all possible influence values, which can result from the outputs of block B. An accumulator, "ACC", then accumulates the influences from the outputs of C, when a pulse signal is received at "IACC".

The circuit cells process a white stone by subtracting influences from the accumulator. This is equivalent to adding negative influences. When a black stone is processed, the "White/Black" signal is set to 0, which enables "ACC" to perform an addition. When a white stone is processed, the "White/Black" signal is set to 1, which enables "ACC" to perform a subtraction.

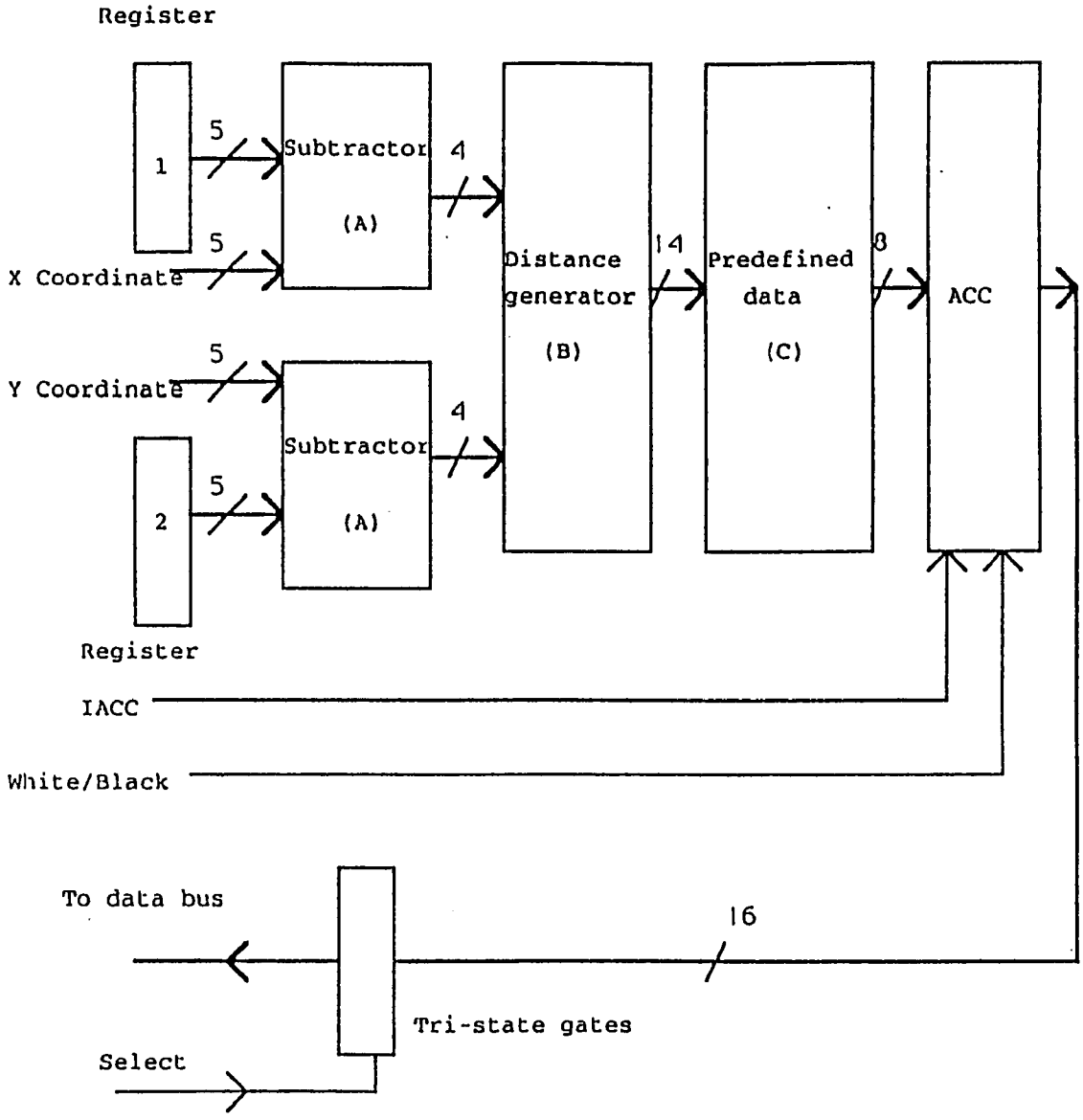


Figure 5.12 The block diagram of the influence calculation circuits.

5.2.2 Simulation and Testing

Figure 5.13 shows the logic circuit we designed to simulate the influence calculation. The circuit is built from registers, decoders, adders, and subtractors. (Subtractors are not available in LOGIMAC™, but are easily constructed from an adder and XOR gates in a straightforward manner.)

Figure 5.13 gives the details of the algorithm: (1) Two 4-bit adder/subtractors are connected to build an 8-bit adder/subtractor. (2) The circuit at bits b5, b6, b7 and a5, a6, a7 converts the 5-bit address into an 8-bit equivalent. (3) After each pair of coordinates are subtracted from the cell's coordinates, only the last 4 bits in the difference are selected as the inputs to the two 3-to-8 decoders. (4) Two 3-to-8 decoders are combined to decode these 4 inputs into 16 outputs for each coordinate. These 16 outputs represent the distances from 7 to -8. Only 11 outputs, -5 through 5, are used which are enough to cover the influence area of a stone. When the distances in either the x or y direction is larger than 5, a value of 0 is generated at the data bus. (5) The outputs of the ten OR gates and two NOT gates placed behind the decoders represent the x and y coordinates of distances from 0 through 5. These outputs are used to activate the tri-state output devices shown in the upper right-hand side of Figure 5.13 and allows the value which is generated by

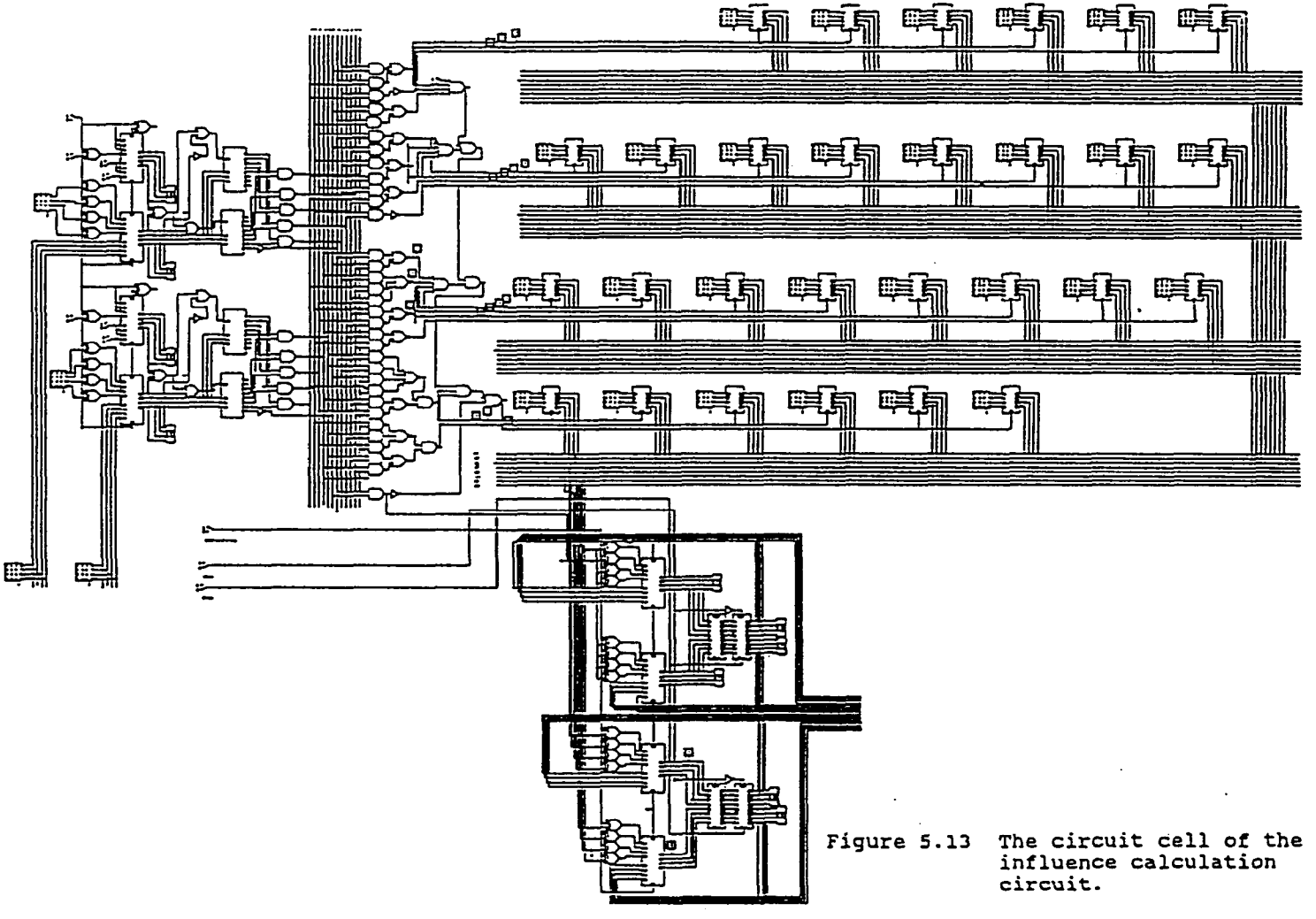


Figure 5.13 The circuit cell of the influence calculation circuit.

presetting two hexadecimal keyboards to access the data bus. (A hexadecimal keyboard is a simulation device in LOGIMAC™ which provides hexadecimal data at the outputs by selecting a hexadecimal number in the keyboard.) (6) If a stone is placed on the cell, a hexadecimal value of 4000 for black stones and C000 for white stones are generated by the circuit. A large positive or negative number in the cell indicates that a stone is in the cell. (7) These values are then summed by the accumulator in the lower part of Figure 5.13, provided input "IACC" is given a pulse signal.

Table 5.1 gives one of our testing results which shows the result in the accumulator when we apply a sequence of coordinates and colors of stones to the circuit given in Figure 5.13. In this test case, registers 1 and 2 are preset to (1,1), (1,5), (5,5), and (10,10).

5.3 The Interference Detector

After a matching point is found, the stones which are matched during the matching process form a joseki pattern. Any stone which is close to the pattern and is not part of this joseki pattern invalidates the joseki pattern. JOSEKI_FINDER discovers the interference stones by generating an interference area surrounding the joseki pattern and tries to find a non-matched stone in the area. (For details, see

Stone position	Stone color	Accumulated influences in ACC (Hexadecimal)			
		preset values for registers 1 and 2			
		(1,1)	(1,5)	(5,5)	(10,10)
(1,3)	BLACK	0022	0022	0003	0000
(2,5)	BLACK	0026	007E	000F	0000
(7,8)	WHITE	0026	007E	0008	FFF9
(10,9)	WHITE	0026	007E	0008	FF9D
(15,15)	BLACK	0026	007E	0008	FF9D
(12,10)	BLACK	0026	007E	0008	FFBF
(2,8)	WHITE	0026	0073	0004	FFBF
(10,10)	BLACK	0026	0073	0004	3FBF
(5,5)	WHITE	0025	006E	C004	3FBF

Table 5.1 Test results of the influence calculation circuit.

Section 3.7.1.) This process is also performed on every node generated by the alpha-beta search for evaluating a joseki pattern. We also designed a hardware circuit to carry out this very task. The circuit generates a result immediately after a stone is loaded into the circuit. The circuit consists of 19*19 cells, with one circuit cell per intersection.

The circuit is designed using the following strategy. For each matched stone, the intersections under the stone's interference area are flagged. Any unmatched stone in a flagged intersection generates an interference. A matched stone flags intersections by sending a 1 to the intersections. These intersections then collect all their incoming signals.

5.3.1 Implementation

Figure 5.14 shows a circuit cell with two functions: one for interference, one for matching. We assume that a stone's interference area covers its surrounding 8 intersections as shown in Figure 5.15. Each cell has both an S-stone and an N-stone. When we detect the interference of a joseki pattern, all N-stones should have been matched with the S-stones in the same cell forming the joseki pattern. Only N-stones generate the interference area necessary to detect nearby interference stones. "N_Exist" is set to 1 when N-stone is not "empty".

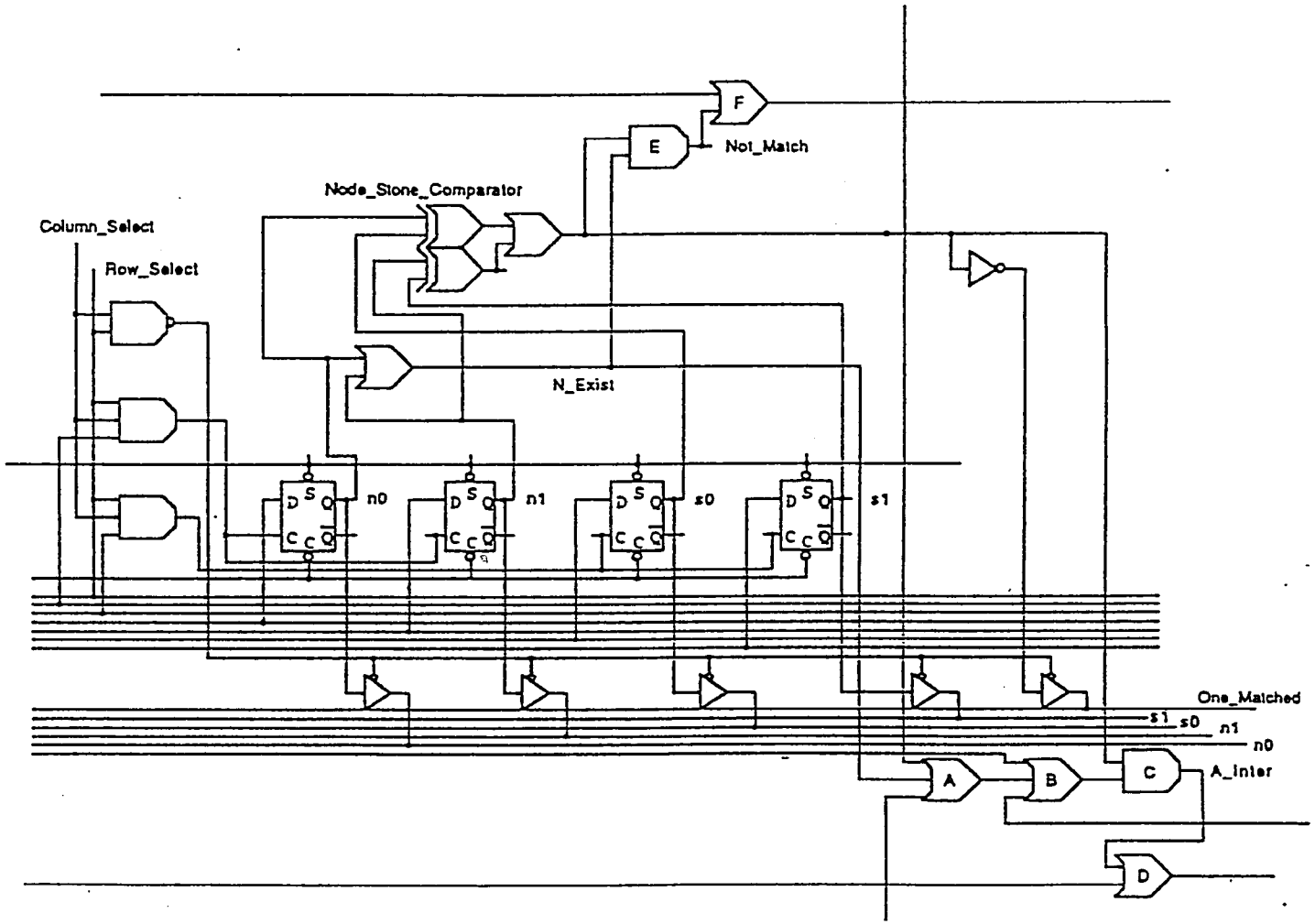


Figure 5.14 The circuit cell used to detect the interferences and to match the pattern.

This signal is then sent to all intersections in the stone's interference area. An OR gate collects the "N_Exist" input signals from the positions which flag the cell. (An input signal of 1 flags the cell; an input signal of 0 doesn't.)

OR gate A collects three "N_Exist" input signals: the upper cell's signal, its own input signal, and the lower cell's input signal. OR gate B collects the results of gate A and the results from three cells in the front and three cells in the back.

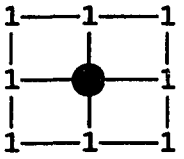


Figure 5.15 Interferences generated by a stone for use in circuit testing.

If both an N-stone and an S-stone are empty or matched, the cell does not generate the interference even if this intersection is set to 1. The condition, S-stone is "empty" and N-stone is not "empty", does not exist. So, only if an S-stone is not "empty" and an N-stone is "empty", can the result of Node_Stone_Comparator be set to 1. Then, if the intersection is also set to 1, "A_Inter" is set to 1. OR gate D is used to collect the "Interference" of the cells in one

row and then OR's them.

5.3.2 Simulation and Testing

Figure 5.16 shows a circuit consisting of 6*6 circuit cells for testing. A 4-input OR gate at the right-bottom position of the figure collects the results from each row and generates an overall interference signal. If the output of the gate, "Interference", is 1, then an interference occurs on the pattern generated by the N-stones.

Table 5.2 shows sequences of S-stones and N-stones placed in the testing circuit and the interference that results.

5.4 The Pattern Matcher

Figure 5.14 also shows how the circuit generates a signal to indicate that the N-stone matches with the S-stone. Iff the N-stone is not "empty", matching the stones is required. The output of gate E, "Not_Match", is 1 when an N-stone is in the cell and it does not match the S-stone. Gate F collects all the "Not_Match" signals for one row of cells. "One_Matched" is 1 if the S-stone and the N-stone in the cell is matched.

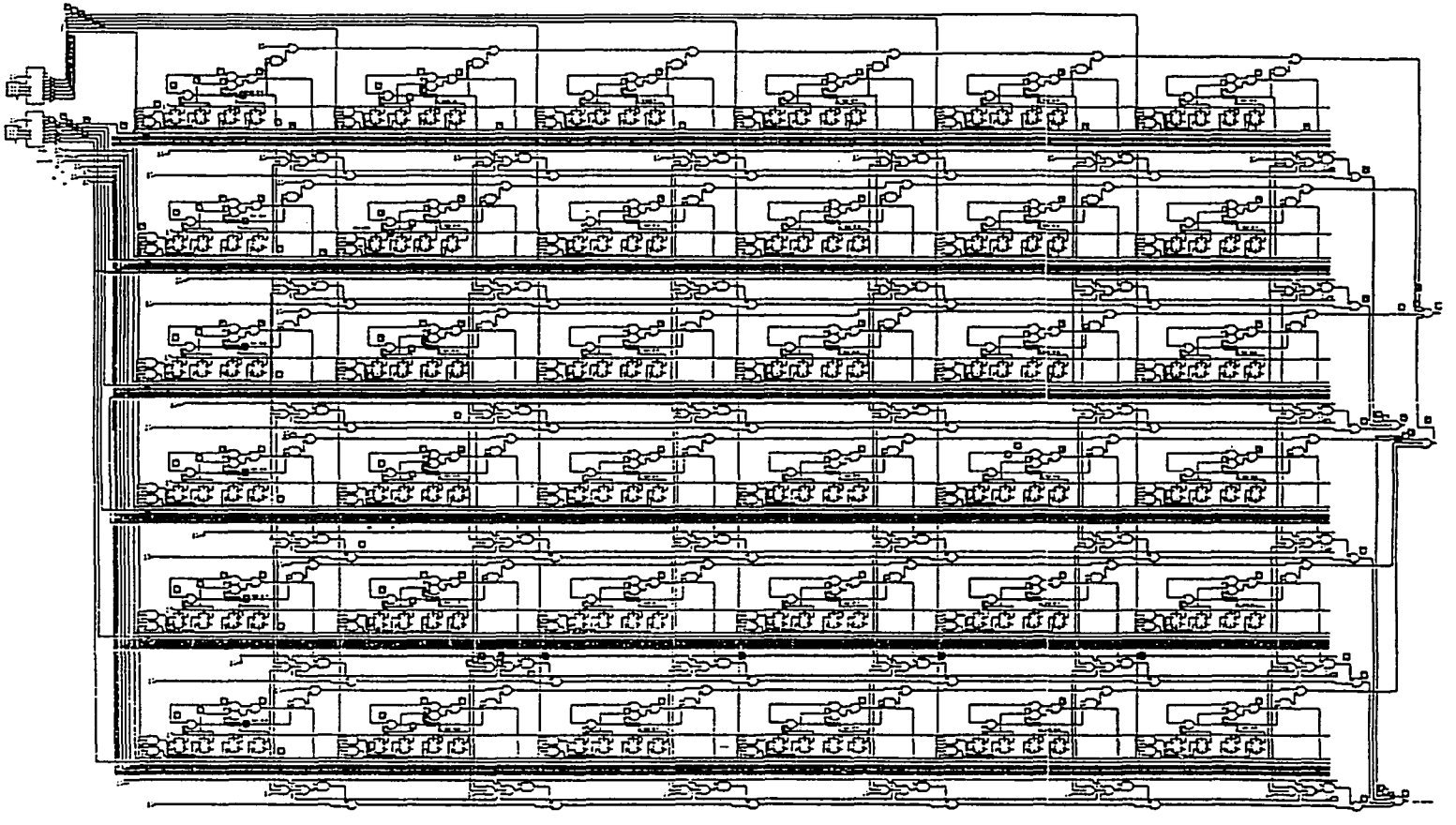


Figure 5.16 The interference detector consisting of 6*6 cells.
The detector is used for testing.

(x,y)	Color	S-Stone or N-Stone	Interference
(1,2)	B	S	-
(3,1)	B	S	-
(4,1)	W	S	-
(5,0)	B	S	-
(3,3)	W	S	-
(1,2)	B	N	0
(3,1)	B	N	1
(4,1)	W	N	1
(5,0)	B	N	0
(3,3)	W	N	0

Table 5.2 Test results of the interference detector.

5.4.1 Simulation and Testing

The 4-input NOR gate shown in the upper right-hand side of Figure 5.15 collects the result of matching the N-Stone and S-Stone in all cells. The output, "All_Matched", is 1 if all N-stones on the game board are matched.

Table 5.3 shows the sequence of stones loaded into the circuits for testing. The table also shows the results of the testing: When all N-stones are matched, "All_Matched" is 1; otherwise, it is 0.

5.5 The Virtual Stone Generator

Remember that before the matching process begins, all S-stones should have been prepared for processing. S-stones give the overall state of the game board. When a matching point is found, all N-stones are matched, but there may be some S-stones which are not matched.

After a matching point is found, alpha-beta pruning is performed. During alpha-beta pruning, we put additional N-stones generated by the move-generation procedure in order to detect whether there are any interferences to the joseki sequence. When the influence evaluation is applied, we

(x,y)	Color	S-Stone or N-Stone	All_Matched
(1,5)	B	S	1
(3,3)	W	S	1
(1,5)	B	N	1
(2,4)	B	N	0
(2,4)	B	S	1
(3,2)	W	S	1
(5,1)	W	N	0
(2,2)	B	S	0
(5,1)	B	S	0

Table 5.3 Test results of the pattern matcher.

consider both the new states generated by the look-ahead procedure and the overall state of the game board. Those N-stones do not have S-stones to match with. Therefore, all N-stones and S-stones, whether or not they are matched, generate influences and virtual stones.

Stones, whether N-stones or S-stones, which are within two positions of any game board border are called *VG-Stones*, stones which can generate virtual stones. The cells which process VG-Stones are called *VG-Cells*. If there are several VG-Stones in one row or column, the one which comes closest to the border determines the placement and the color of the virtual stone. (For details, see Section 3.7.2.) Figure 5.17 shows how the circuit generates a virtual stone. Stones at lower positions have higher priorities. VG-Stone0 is the stone on the border; it has the highest priority. The virtual stone generated by these VG-Stones are stored in the two D flip-flops shown at the bottom of the figure.

For the cells which handle the stones on the border or one position away from the border, two AND gates and two OR gates are added. For VG-Cell0, when there is a stone in the cell, the output of gate C is 0. A 0 in this output disables the propagation of the inputs, i1 and i2, and sets a1 and a2 to 0. The outputs of gates B1 and B2 are then the color of the stone in VG-Cell0. (01 represents BLACK, while 10 represents WHITE.)

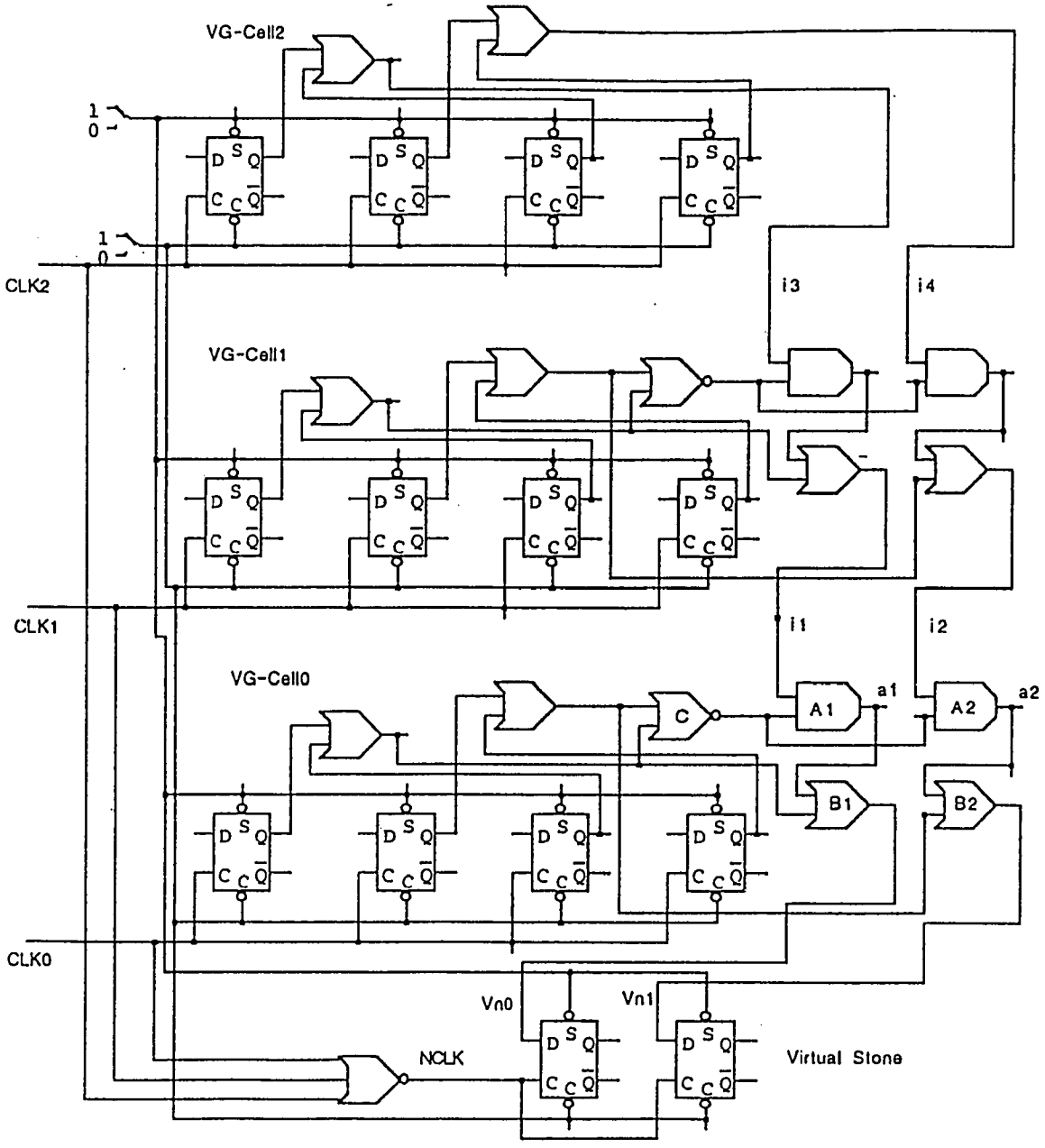


Figure 5.17 The virtual stone generator.

When there is no stone in VG-Cell0, the outputs of gates B1 and B2 equal i1 and i2, respectively. The inputs i1 and i2 are then propagated through VG-Cell0 and reach the cell for the virtual stone. VG-Cell1 performs in the same manner on inputs i3 and i4. VG-Cell1 and VG-Cell2 allow the input signals, i1 and i2, or i3 and i4, to propagate through the cells if they are empty. However, if there is a stone in the cells, the cells generate the color of the stone as output and ignores the input signals.

5.5.1 Simulation and Testing

In LOGIMAC™, data are loaded into D flip-flops at the rising edge of input signal C. Therefore, a VG-Stone is loaded at the rising edge of input signal "CLK". Vn0 and Vn1 are generated after a VG-Stone is loaded into their D flip-flops and the output of the flip-flops from Q are then propagated through several logic gates to the cell of the virtual stone. The virtual stone cannot be loaded at the same time as that of the VG-Stones because of these propagation delays. A 3-input NOR gate is added to the circuit, with output "NCLK" inverting all clock signals. When the inputs "Vn0" and "Vn1" are stable, i.e., the clock signals are at a falling edge, then a virtual stone should be loaded in synchrony with the "NCLK" signal. (The "NCLK" signal is activated whenever any VG-Stones are loaded.)

In the simulation, we loaded different stones in the VG-Cells as shown in Figure 5.17. The virtual stone always follows the stone which has the highest priority.

5.6 A Discussion on the Design of a Joseki Processor

Figure 5.18 shows the block structure of a joseki processor. The controller fetches the information from the joseki tree and controls all the circuits. The controller can be implemented as part of the host machine or as a custom-designed microprogrammed processor. It executes the algorithm we developed, Procedure 3.1, but leaves all the static evaluations - killing detections, stone matchings, interference detections and influence calculations - to the hardware circuits.

Procedure 5.1 shows the algorithm executed by the controller. In the algorithm, all variables in double quotes are signals read from the hardware circuits. All operations are performed at N-stone positions. Before the algorithm is executed, all S-stones are on the circuits and ready for processing. The controller sends the x and y coordinates and the color of the node to all circuits, and receives "All_Matched", "One_Matched", "Interference" and "Killing" signals from the hardware circuits. The controller marks the killing node, traverses the tree, decides which node to load

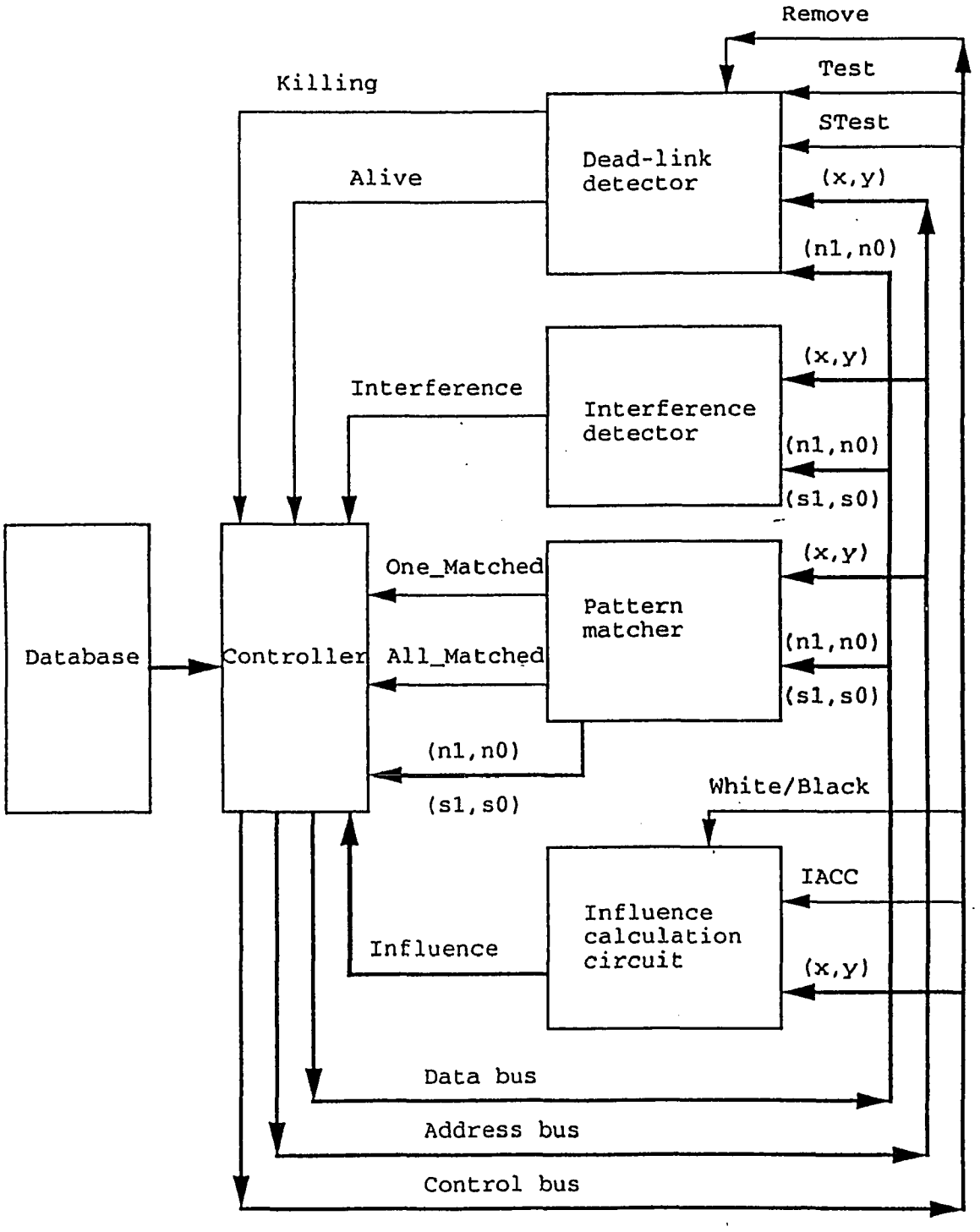


Figure 5.18 Block diagram of the joseki processor.

```
Match_a_Node(Node)
{ get (x,y,color) of the Node from the joseki tree;
  load Node to the N-stone of cell (x,y);
  if "Killing" is 1,
       {
        push all stones onto the stacks;
        mark the node "KILL";
        remove dead-link(s);
       }
  if "One_Matched" is 0,  /* a mismatch */
      {
       if Node is not a be-killed node,
       /* if Node is a be-killed node, do nothing */
              if Node is marked as "KILL",
                  pop all stones from the stacks;
              else remove Node;
              return;
       }
  else /* this is a match case */
       {
         /* match next level */
         define SonNode = the first son of Node;
         while (SonNode exists)
              {
                Match_a_Node(SonNode);
                SonNode=The brother of SonNode;
              }
         if all SonNodes fail to match,
              {/* some be-killed nodes mismatched */
                while "All_Matched" is 0
                     {
                       remove Node; /* adjust matching point */
                       Node=Node's parent node;
                     }
                if "Interference" is 0,
                   alpha-beta-evaluation(Node);
              }
       }
  if Node is marked as "KILL", pop all N-stones;
  else remove Node;
}
```

Procedure 5.1 The algorithm for the controller to process a node.

onto the circuits, and uses alpha-beta pruning. Then, after the influences of an intersection are calculated, the controller has to perform a dot product weighting the influences. The remainder of the processing is performed by hardware circuits.

CHAPTER 6

Conclusion

This thesis has presented algorithms to process josekis, a parallel algorithm to calculate the influences of the intersections, and the logic circuits needed to provide the computational power for a Go program. After many cycles of design development, and enhancement, our joseki processing algorithms are capable of handling a large opening-game database in a short time.

6.1 Contributions

The main features of our joseki processing algorithm are that we are able to find a joseki pattern from a non-joseki sequence and process them rapidly. Also, because we introduce the influence evaluation into the algorithm, the joseki processing is no longer restricted to local computation.

When additional joseki and non-joseki sequences are added to the opening-game database, retrieving the needed information from the database becomes the processing bottleneck. We solve the problem by separating the joseki tree into subtrees and use the cap tree consisting of the

nodes in the top four levels of the joseki tree and identifying which subtrees are to be fetched. This avoids unnecessary I/O accesses and allows our program to process a large opening game database. After a joseki pattern is found, alpha-beta pruning and influence evaluation are applied to find a joseki based on the overall state of the game board.

The key feature of the parallel influence calculation algorithm and the logic circuits we designed is that *all intersections are distributed to all processors*. The parallel influence calculation algorithm retrieves all the influences on an intersection generated by stones, from a matrix and accumulates them. With the iPSC-D5, because the number of processors is limited to 32, the processors must be carefully and evenly distributed. Distribution methods for this problem have not been researched heretofore, and we tried several different methods, before settling on the tiling described in Chapter 4.

In designing the dead-link detector, influence calculation circuit, interference detector, and pattern matcher, a single circuit cell handles just one intersection. Our circuits detect if an intersection is alive, is interfered with, or is matched. The influences are calculated in parallel using an intersection as the unit. These hardware circuits evaluate

the conditions on the game board so rapidly that results are reported immediately after a stone is loaded.

In summary, the contributions of the thesis are:

(1) We designed the first hardware circuits for Go programs. Our dead-link detector discovers a dead link or a suicide move. Our pattern matcher finds a defined pattern in the game board, and our interference detector finds if any stones exist around the defined pattern. Our influence calculation circuit accumulates the influences of all intersections.

(2) We designed and implemented algorithms to recognize joseki patterns on a Go game board and to select a joseki from a joseki tree based on the overall state of the game board. Selecting a joseki is no longer restricted to local computations.

(3) We developed a parallel algorithm in a distributed environment on the iPSC-D5 hypercube machine. Several methods in distributing 32 processors to 19*19 intersections were tried; the most efficient was selected.

(4) We used a cap tree to identify the needed subtrees. This removes the bottleneck that has plagued joseki processing, and allows a joseki tree or an opening-game database to be extended with only a slight increase in execution time.

(5) We introduced the concept of virtual stones. The influences of an intersection now more faithfully reflect the degree of control over an intersection by either BLACK or WHITE.

(6) We developed a special-purpose pattern recognition procedure to recognize stone positions and sequence numbers on stones in a joseki dictionary.

6.2 Future Research Directions

Suggestions for future research in the areas examined here include:

(1) The end game has very similar properties to the opening game; we might use the same approach to process the end game.

(2) Extend the program and design circuits to handle the middle game. Handling the middle game requires more

heuristics and algorithms, and more computational power.

(3) Playing Go games depends critically on proximity perception. A good Go program would be able to apply AI work on proximity perception.

REFERENCES

[1] Feng-hsiung Hsu, Thomas Anantharaman, Murray Campbell and Andreas Nowatzyk, "A Grandmaster Chess Machine", *Scientific American*, 263:44-50, October 1990.

[2] Feng-hsiung Hsu, *Large Scale Parallelization of Alpha -Beta Search: An Algorithmic and Architectural Study with Computer Chess*, Ph.D. Thesis, Carnegie-Mellon University, CMU-CS-90-108, February 1990.

[3] Edward Lasker, *GO and GO-MOKU THE ORIENTAL BOARD GAMES*, Dover, New York, 1960.

[4] Bruce Wilcox, "Issues in Joseki Processing", *Computer Go*, No.4, pages 18-23, Fall 1987.

[5] Keh-Hsun Chen, "A Data Structure for a Joseki Dictionary", *Computer Go*, No.6, pages 17-18,24, Spring 1988.

[6] Ander Kierulf, Ken Chen, and Jurg Nievergelt, "Smart Game Board and Go Explorer: A study in software and knowledge engineering", *Communications of the ACM*, 33(2):152-166, February 1990.

[7] David W. Erbach, "The 1990 North American Computer Go Championship", *Computer Go*, No.14, pages 3-5, Spring /Summer 1990.

[8] A.L. Zobrist, *Feature Extraction and Representation for Pattern Recognition and the Game of Go*, Ph.D. dissertation, University of Wisconsin, 1970.

[9] J.L. Ryder, *Heuristic Analysis of Large Trees as Generated in the Game of Go*, Ph.D. dissertation, Stanford University, 1971.

[10] Walter Reitman and Bruce Wilcox, "Perception and Representation of Spatial Relations in a Program for Playing Go", *Computer Games II*, pages 192-202, Springer Verlag, New York, 1988.

[11] Walter Reitman and Bruce Wilcox, "Pattern Recognition and Pattern-Directed Inference in a Program for Playing Go", *Computer Games II*, pages 214-233, Springer Verlag, New York, 1988.

[12] Walter Reitman and Bruce Wilcox, "The Structure and Performance of the INTERIM.2 Go Program", *Computer Games II*, pages 234-247, Springer Verlag, New York, 1988.

[13] Keh-Hsun Chen, "The Move Decision Process of Go Intellect", *Computer Go*, No.14, pages 9-17, Spring /Summer 1990.

[14] Elwyn R. Berlekamp, "Introductory Overview of Mathematical Go Endgames", *Proceedings of Symposia in Applied Mathematics*, 43:73-100, August 1990.

[15] Ishida Y., *Joseki Dictionary*, Volume 1 and 2, World Journal Publication, Taiwan, 1982.

[16] *iPSC™ System Overview Manual*, Intel™, November 1986.