1235

A Go program which uses a new power system

THÈSE

présentée à la Faculté des Sciences pour obtenir le grade de docteur ès sciences par

Xuedong LUO

Novembre 1994

UNIVERSITÉ DE NEUCHÂTEL

Institut d'Informatique et d'Intelligence Artificielle

Rue Emile Argand 11
2007 Neuchâtel
Suisse

IMPRIMATUR POUR LA THÈSE

Equilibrator, a Go program which uses a new

power system

de M. Xuedong Luo

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

Messieurs P.-J. Erard, H.-H. Nägeli, J.-P. Müller,G. Coray (EPF-Lausanne) et K. Chen (South Carolina University)

autorise l'impression de la présente thèse.

Neuchâtel, le 11 novembre 1994

Le doyen:

H.-H. Nägeli

We do not weigh money to know its value

To my wife, Yanping, for her love, patience
and the last moments we could have shared

Acknowledgements

The work reported here has been completed in the Computer Science Department of the University of Neuchatel in Switzerland.

I would like to express my gratitude to my thesis advisor, Pierre-Jean Erard, for his guiding, supporting and encouraging which made this work possible. I am grateful that he gave me the opportunity to create Equilibrator. Thanks are also go to Hans-Heinrich Nägeli, Ken Chen, Giovanni Coray and Jean-Pierre Müller for their reading and commenting on this thesis.

Finally I want to thank my family and friends, very especially Elisabeth Erard and Pontien Déguénon, for their encouragement, support, friendship and love that I have experienced during the past years.

Table of Contents

ABSTRACT

Computer Go is a AI (Artificial Intelligence) area which attracts great attention from AI researchers.

Go is a complex system which is more similar to the real world than other games. Studying Go can help us to simulate the real world with a computer formal system and to find the method to solve the problems in the field of AI.

In this thesis, the author corrected some concepts and invented new concepts in Computer Go. In particular, a new power (influence) system is introduced and analysed. Power System is based on a function which expresses the influence of a stone on to the other parts of the play. This function, a result of an analogy with the physical world, has an universal application field. It can be used to evaluate a configuration of the board, to define group membership and also to find the exits of the group. This new power system is natural, simple and more efficient than what was used up until today. The new Power System presented by the author can be seen as a general method to represent a situation and to solve problems which occur in the real world. This generalisation will improve the resolution of several important problems of AI.

Equilibrator is a computer program which implements this method. Equilibrator has a global view on the board and plays game strategically rather than tactically. In consequence, it plays sometimes better than other programs (when the strategy points are more important than other points for the situation at that time) and sometimes worse than others (when the local points are more important than strategy points for the situation at that time).

1. GO

1.1 HISTORY OF GO

Go is a Chinese strategy game*. It is the most ancient, the most complicated, the most interesting and the most attractive game in the world. With more and more people starting to play, there are currently more than 30 million Go players world-wide.

Who invented Go and when was it invented ?

One of the explanations (popular explanation) is: About 4000 years ago (2300 B.C.), two of the early Chinese rulers Yao and Shun may have invented Go to develop human intelligence and morale.

Another explanation (precise explanation) is: About 3000 years ago (1200 B.C. - 1000 B.C.), Go was invented by Ji-Zi "箕子"one of earliest Chinese astrologers and pre-creator of Yi-Jing (YiKing) and Taoism, in Shan-Xi province of China (the stones of Go are called Qi-Zi "棋子" in Chinese, perhaps after this man). This second explanation was given by Mr. YANG Xiao-Guo, assistant research fellow of the Social Science Institute of Shan-Xi province, in early 1993.

About 2200 years ago (220 B.C.), Go was exported to Korea. During the Tang Dynasty (618 - 907 AD.), the Japanese imported et adopted Go. The first official game between China and Japan was held in the capital of China, in 854.

From the birth of Go until the beginning of the Qing Dynasty, spanning 2500 years, China was one of the most economically strong countries and the strongest "Go country" in Asia. But since then, China's economy has suffered, and Japan took its place as the best "Go country". Japan took the Go rules of Tang Dynasty but made some important improvements and developments.

Now, there are about 10 million Go players in China and as many in Japan and South Korea. Europe imported Go from Japan in the 19th century, but it is not yet a popular game in Europe.

---

* here: "game" means a game using intelligence, not physical movement.

The following table displays the spread of Go throughout the world, especially the number of players in certain countries and the thousandth proportion to the population in 1989.

Country	Club	Player	‰
U.S.A.	132	80,000	0.32
Germany	138	46,000	0.60
USSR	80	40,000	0.14
Great Britain	59	35,000	0.61
Netherlands	33	20,000	1.43
Brazil	8	20,000	0.13
France	48	15,000	0.27
Canada	22	7,000	0.27
Yugoslavia	15	6,000	0.25
Czechoslovakia	19	4,000	0.25
Romania	25	4,000	0.17
Austria	13	3,000	0.42
Poland	9	3,000	0.08
Switzerland	9	2,000	0.33
Denmark	4	1,000	0.20
Norway	8	1,000	0.25
Hungary	3	1,000	0.10
Argentina	3	1,000	0.03
Belgium	6	800	0.08
Sweden	6	800	0.10
Italy	5	800	0.01
Spain	5	800	0.02
Finland	5	500	0.10

1.2 MATERIAL OF GO

Go is written 圍棋, pronounced Wei-Qi in China,
written 圍棋, pronounced Wei-Chi in Taiwan,
and written 碁, pronounced Igo in Japan.

Go was simplified from "Igo" in order to be easily adopted by the West.
Go-ban is the board on which we place the stones. See Figure 1.1

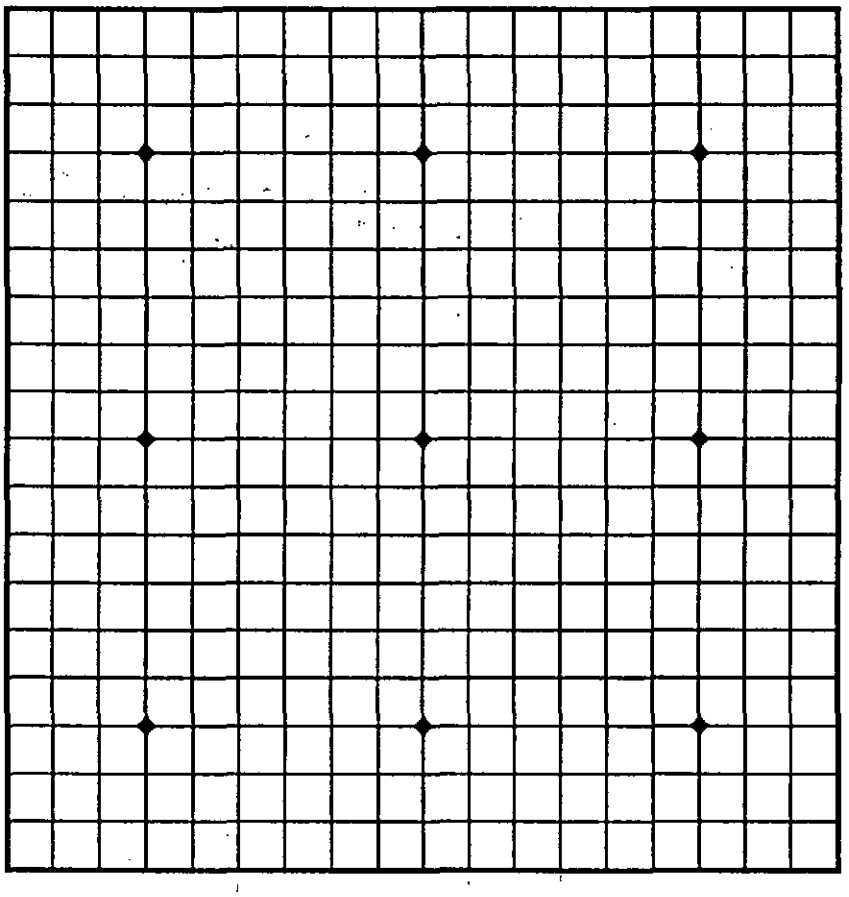


Figure 1.1: Go-ban and stones: ○ ●

Go-ban has 19 x 19 = 361 intersections which represent the number of days in one year (don't forget that Go-ban was designed 3000 years ago!). It has 9 "Xing" (stars) which represent 9 planets. Among them, 5 stars divide Go-ban into 4 parts which represent the 4 seasons. The shape of Go-ban is a square which represents the earth. The shape of the stone is a circle which represents the sky. The center star is called Tian-Yuan (center of the universe); it is also called "Tai-Ji" which is an important concept in Tao. The lines of Go-ban are called Tao (path). The black and white stones represent Yin and Yang. Go is not only a tool of astrologers, but can also be used by Taoists to study Yin-Yang.

1.3 RULES OF GO

There are three sets of rules for Go: Chinese rules, Japanese rules and Ing's rules (Mr. Ing is a Taiwanese businessman). The differences between them are small and they give the same results (the winner) in most cases. Even so, there are some differences: Japanese rules originate from the old Chinese rules, they are more complicated, but less complete. Chinese rules are simpler. Ing's rules are both complete and complicated.

We introduce the Chinese rules as follows, the simplest to understand and to implement into a program:

(1) Go is a game with two players. Each player chooses a color of Go. Black plays first and the two players play alternately by placing a stone of their color at an empty intersection (point) on the 19 x 19 grid Go-ban. The stones cannot be moved from one point to another. A player can pass at any time.

(2) A **play** (a move) involves one player taking a stone of his color and putting it at an empty point on the Go-ban.

At the end of the game, the aim is to control more board points than the opponent .

(3) Stones of the same color connected horizontally or vertically by grid lines belong to the same **block**.

Figure 1.2 shows three examples of blocks.

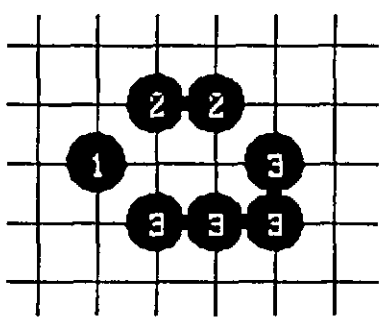


Figure 1.2: Three blocks 1, 2, 3

(4) A **liberty** of a block is an empty point adjacent to a stone of the block. The stones of one block have the same liberties, and they "live" or "die" together. (see rule 5).

Figure 1.3 shows an example.

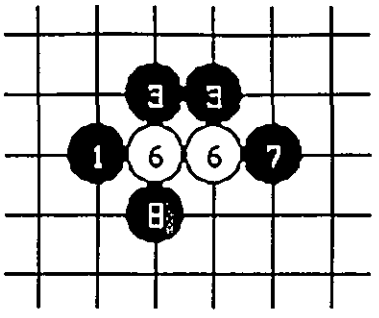


Figure 1.3: Block 1,7,8 have 3 liberties, block 3 has 4 liberties and block 6 has 1 liberty

(5) A block is **dead** when it has no liberties. Whenever a block is dead it has to be removed from the Go-ban. If a player places a stone which simultaneously takes away the last liberties from the blocks of both colors, only the opponent blocks are removed from the Go-ban.

Usually a dead block is called a "captured" block. I renamed this definition because I think the word "captured" is incorrect. Figure 1.4 shows how to kill a block.

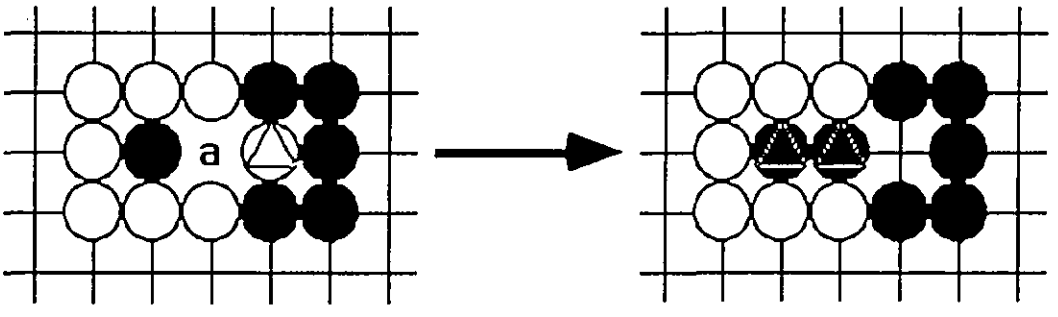


Figure 1.4: Killing a block

After Black plays a stone at point *a* on the board, the white block △ is dead and removed from the board; even if the block ▲ has simultaneously no liberties with the block △.

(6) A **white (black) forbidden point** is an empty point in which the white (black) stones would die if white (black) plays at this point. It is forbidden for white (black) to play at a white (black) forbidden point.

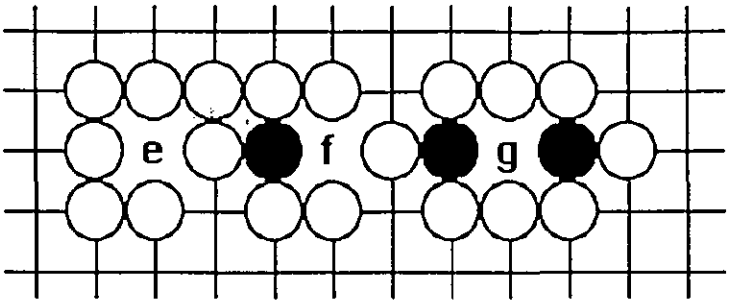


Figure 1.5: The points e, f, g are black forbidden points

This rule means that suicide is not permitted. The points e, f, g are also called "**eyes**" of White. This prohibition does not exist in Ing's rules and I agree that it is unnecessary.

(7) It is forbidden to generate the same situation more than once during the same game.

See Figure 1.6 and Figure 1.7.

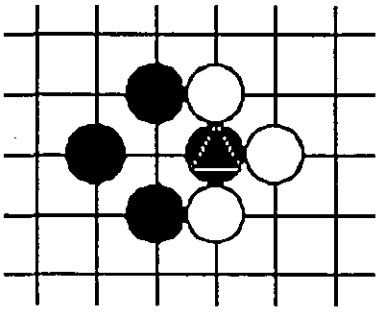


Figure 1.6: White to play

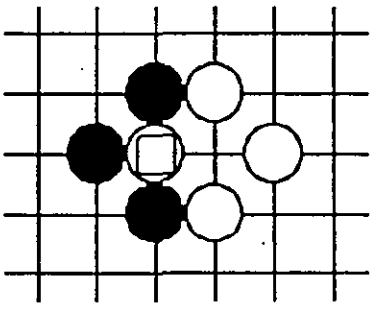


Figure 1.7: Black to play

After White plays ◯ and kills ● (Figure 1.7), Black cannot play a black stone at the ● point of Figure 1.6 and kill ◯ in the next play. Black has to play elsewhere on the board to avoid the situation occurring before play at ● point of Figure 1.6.

(8) The game finishes when the two players agree on the ownership of each point (empty and occupied) in the Go-ban.

(9) White receives the equivalent of 2.75 stones at the end of the game.

An additional 2 stones to White is considered insufficient and an additional 3 stones to White is considered too much; the number 2.75 was chosen to avoid a draw.

(10) Count the both players' possessive points. The player who possesses more than 180.5 stones wins.

These Chinese rules are used only in China (PRC); Japanese rules are used in Japan, Europe and U.S.A.; Ing's rules are used in Taiwan and in the tournaments sponsored by Ing's Fund.

1.4 THE RATING SYSTEM OF GO

The rating system is tied to the handicap system: one rank difference is equivalent to one handicap stone. Handicap stones are placed on the board as the Black's first move (the weaker player always takes black).

The ranks are expressed in **kyu** and **dan**, as illustrated by the scale as follows:

strong < ----------> weak
Professional dan : 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1
Amateur dan : 6 - 5 - 4 - 3 - 2 - 1
Amateur kyu : 1 - 2 ... - 25

One rank difference is equivalent to one handicap stone for an amateur player.
For example, a 2 amateur dan gives two stones to 1 kyu and twenty one stones to 20 kyu.

This is an official classification. Some players who are stronger than the majority of 9 dan players are called "Super-players", and the best among them are called "Hyper-players". To receive the classification promotion, the player must pass the yearly competition specifically for the classification promotion. Some good players do not have enough time to take part in this kind of competition (this competition is not financially interesting), so a Super-player or a Hyper-player is not necessarily 9 dan. (S)He could be 9 dan, 8 dan, 6 dan, even 3 dan !

Almost all Hyper-players had learned to play Go before they were 10 years old. There are about 2-3 Hyper-players in China, 3-4 Hyper-players in South Korea and 6 -7 Hyper-players in Japan.

1.5 THE DIFFICULTY OF GO

Go is a very complex game. Even for a good player there are 10 ~100 possibilities to consider for each play. Because of that, a good Go player needs high calculation capabilities, good look-ahead abilities and strategic foresight. Choosing a move is very difficult. We often hear the dialogue between two hyper-players as follows:

Hyper-player A: "I think that black is favoured now."
Hyper-player B: "No. I think that white is favoured now."

Hyper-player A: "I think that Black should play this point now."
Hyper-player B: "No! Never! It is the worst point for black, I would never play black at this point!".

Hyper-player A: "Why didn't you play here ?".
Hyper-player B: "Oh yes! I totally missed that possibility".

Before becoming a 1 amateur dan (a respectable amateur rating), one should play about several hundreds or a thousand games. But not everyone who has played a thousand games will become a 1 amateur dan.

2. COMPUTER GO

2.1 HISTORY OF CGP (COMPUTER-GO-PLAYING)

In 1968, the first Go Program, written by A. Zobrist for his Ph.D. at the University of Wisconsin, can play a complete game of Go [Zobrist 70]. Even when this program is considered to be a "complete novice", it is a landmark in the history of CGP.

In 1972, J. Ryder designed the second Go Program for his Ph.D. at Stanford University.

From 1972 to 1979, W. Reitman and B. Wilcox worked on their program INTERIM.2. Later, Bruce Wilcox (considered to be the pioneer of Computer-Go) improved this program and named it NEMESIS. NEMESIS is the first Go Program to play Go "reasonably" well and have been commercially successful. It was classified as 20 kyu by the American Go Association in 1984.

Since 1985, Mr. Ing offered a prize of about 1.5 million dollars to the inventor of the first Go Program which could defeat a Go player of his choice before the year 2000. He is also a strong Go player himself and has successfully used the strategies of Go in business. Mr. Ing is the sponsor of several important annual computer Go tournaments and also the sponsor of the "Ing's Cup International Go Tournament". This tournament is one of the earliest and the biggest professional Go tournaments in the world (a total prize of one million dollars for the tournament). Thanks to Mr. Ing, about twenty Go Programs have been written since 1985, and several yearly computer Go tournaments have been held in the world, the most important being: International Computer Go Congress; Computer Olympiad Conference on Computer Game; North America Computer Go Championship.

The following is a summary of top winners of last three years:

International Computer Go Congress:

	1st	2nd	3rd
1991	Goliath	Go Intellect	Dragon
1992	Go Intellect	HandTalk	Goliath
1993	HandTalk	Star of Poland	Go Intellect

Computer Olympiad Go Conference:

	1st	2nd	3rd
1990	Go Intellect	Swiss Explorer	Go 4
1991	Goliath	Go Intellect	Explorer 90
1992	Go Intellect	Go 4.3	Archmage

(There was no Computer Olympiad in 1993)
The best program is estimated about 9 kyu.

Programming Go is harder than it looks! In the early 80's, the Japanese wanted to produce a shodan (1 amateur dan) Go program in their "Fifth Generation Project". Several years later, this sub-project was abandoned because it was considered impossible to achieve. We will show in the section 2.3 why it is difficult to write a good Go program.

2.2 SEVERAL IMPORTANT CONCEPTS OF GO

2.2.1 Group

"Group" is an important concept and is useful for the Go player, but its definition is problematic. Figure 2.1 shows some groups recognized by the Go player.

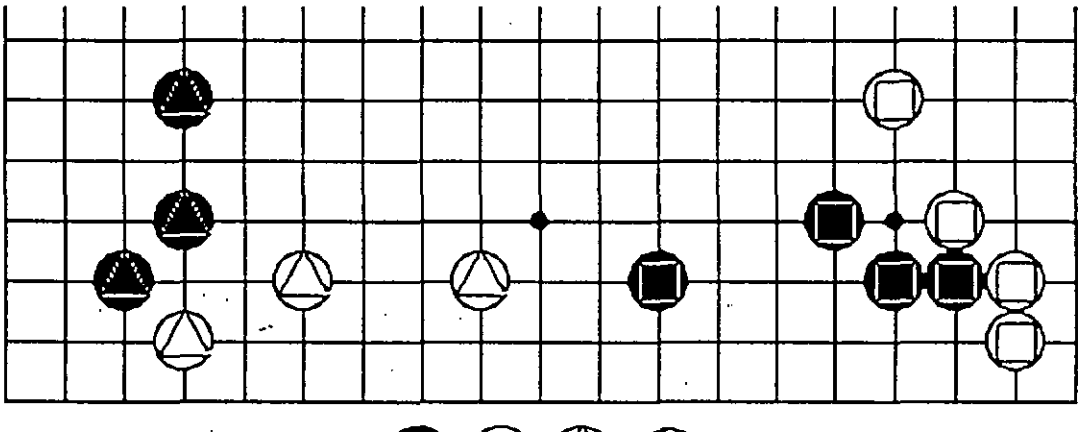


Figure 2.1: Four groups , , , recognized by the player.

Usually, the group definitions are like these: "A group is a fighting unit which consists of blocks of the same color that live or die together"; "A group is a set of blocks that typically fight together"; "A group is a collection of blocks forming a strategical unit". These definitions are easy to understand by a human, but not by a program. I call this kind of definition "**functional definition**". I also suggest another definition (I call it "**characteristic definition**") so that the program can recognize the group by this definition:

Two stones of same color having a "strong relationship" belong to the same **group**.

The "strong relationship" design fitting this definition will be done in section 3.3.

2.2.2 Life and Death (LD)

Go is a game in which two players intend to control more board points (territory) than the opponent. As the board is limited, killing the opponent groups has a double effect: an increase of the player's territory and a reduction of the opponent territory. In this sense, Life and Death (LD) problems play a key role in Go, but they are difficult for both the player and the program. During each game the player must determine whether a given group should live or die. The LD Problem detects whether a given group is captured, able to be captured or alive. Knowing the exact status of a group is important, since groups that lack space for two eyes are weak and are the attack targets for the opponent. Playing a stone in a LD fight which does not threaten to revive a captured group or kill a living group is wasteful.

I will give several new important definitions as follow:

A group is **captured** when it has not enough space to make at least two eyes, otherwise it is **alive.**

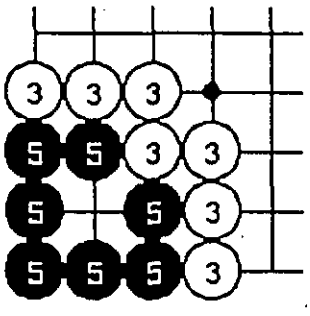


Figure 2.2: Black group 5 is captured

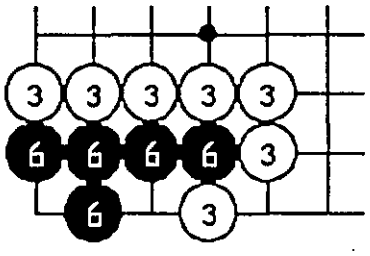


Figure 2.3: Black group 6 is captured

A group is **CAPN** (Captured After Playing N stones) if it can be captured after the opponent plays N consecutive stones. CAP∞ means the group is alive eternally.

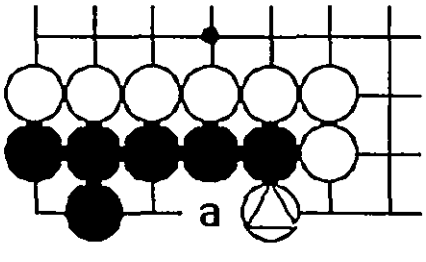


Figure 2.4: Black group is in a critical situation and is CAP1.

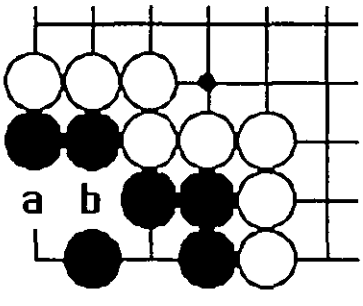


Figure 2.5: Black group is CAP2.

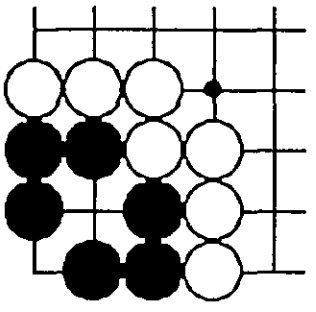


Figure 2.6: Black group is CAP∞.

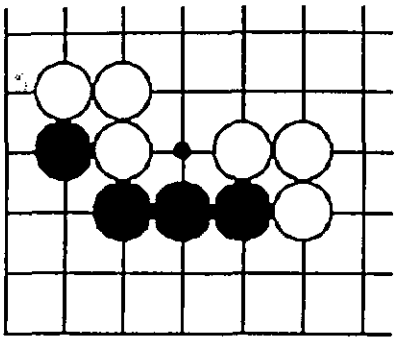


Figure 2.7: Black group is CAP1.

In Figure 2.2 and 2.3, the black groups are captured, so they are CAP0. In Figure 2.4, the black group is in a critical situation; if White plays a stone at point *a*, then the black group is captured; so the black group in Figure 2.4 is CAP1. The black group showed in Figure 2.5 is CAP2, because if White plays at points *a* and *b*, then the black group is captured. But if White plays at *a* or *b* and Black plays at *b* or *a*, then Black is not captured. The black group of Figure 2.6 is eternally alive, because it has two eyes and White can never kill it. The group living eternally is defined as CAP∞. The example in Figure 2.7 is more delicate. Ever for a "dan" player, it is not easy to determine the situation of the black group. The black group is CAP1.

We can get some interesting results from the discussion above:

"Group is CAP0" means: the group is captured.
"Group is CAP1" means: the group is in danger (weak).
"Group is CAPN ($2 \leq N < \infty$)" means: the group is alive (safety).
"Group is CAP∞" means: the group is eternally alive.

CAP1 groups are often the opponent attack targets and so are CAP2 groups sometimes. For example, in Figure 2.4, before White plays △, the Black group is CAP2 and it is safe, but after White plays △, Black becomes CAP1 and unsafe. After that if Black plays at point *a*, then it is CAP∞; otherwise, if White plays at point *a*, then the black group is CAP0. Finding the status of a group (CAP0, CAPN, CAP∞) needs a tree search process (see section 2.7 and 3.6).

2.3 HOW GO PROGRAMS WORK

2.3.1 General Problems for the Go Player

To know how a Go program works, let us see first how a Go player chooses his (her) candidate point from the board. The Go player should take the following steps to make his (her) decision.

(I) Could I follow the standard corner opening plays (joseki see section 2.4) ? If yes, follow it.
(II) Are there weak groups ? If not, select a point to make own territory or destroy opponent territory.
(III) If I have a weak group, should I protect it ? If yes, select a point to protect it.
(IV) If my opponent has a weak group, should I attack it ? If yes, select a point to attack it.

This is a *sketch* of how a Go player chooses candidate points but only a sketch. Because Go is a game of slowly acquired position advantages. The objective of Go is to control more territory than your opponent, but there are a multitude of direct and indirect ways to accomplish that goal. One may attempt to make territory, attack opponent weak groups, protect own weak groups or destroy opponent territory. There is no central point on which the game must focus, so players may concentrate their energies wherever they please. The size of the board causes the game to fragment into many small battles, giving no unique focus. Tactical gains do not outweigh position ones, nor do various position gains seem to have an unifying theme.

To realize these strategy steps, the Go player has to:
(a) have a large knowledge about standard corner opening plays and know how to use them.
(b) identify the groups status: captured, weak, alive.
(c) select a point to play, depending on his (her) judgement of the situation on the board.

To realize point (a), a Go player must work hard during several years (there are more than 30,000 standard corner opening plays).

To realize point (b), a Go player must be able to identify a group and its status. The player has to do a local look-ahead search to identify the group's status (Life and Death. See section 2.7). It is difficult to identify a group status correctly; even a Hyper-player can make a mistake in doing it.

To realize point (c), the player should do a look-ahead search and evaluate the situation on the board for selecting a point to play. Selecting an attack point or a defence point is more difficult than identifying the groups' status. It does not only depend on the group status, but it also depends on the player's strategy and his (her) experience.

2.3.2 How Go Programs Handle These Problems

Two major difficulties of computer playing are how to deal with a great amount of information (i.e. the 30,000 known opening plays in Go) and the art of reasoning needed to win the play. For the first one, a good system of storing, retrieval and pattern matching is needed. For the second one, there are historically three trends which can be combined together:

- exhaustive search; generally this leads to an enormous and almost unmanageable number of cases (combinatorial explosion) and also to rather an unrealistic implementation.
- use of heuristics, which can bring a surprisingly good but also a catastrophic solution.
- use of abductive processes, which help to choose a good strategy among various possibilities.

A Go-play consists of heterogeneous parts, which aim to occupy the board in the best possible manner and are broken by the special attack phases in order to kill the opponent group. Consequently Go playing requires a hybrid strategy whose elements are:

- Test of corner situation with a pattern matching based on a pattern library. (see section 2.4).
- Evaluation of board situation with a special influence function. (see section 2.5).
- Identification of groups and evaluation of their status (see section 2.6).
- If there are weak groups, then selection of candidate attack-defence points, application of an α–β look-ahead based on life and death process to these points (see section 2.7).
- Capture of the group, ladder phase (see section 2.8).

Generally, programs choose their candidate points from following point generators:

- pattern matching.
- attack opponent weak groups.
- protect own weak groups.
- make own territory.
- destroy opponent territory.

Like each generator can give some candidate points to achieve a goal, programs give each point some goal-priority value depending on which generator it comes from. There are three fundamentally different ways to decide which point should be played:

- Goals-priority

  Programs scan a list of goals to achieve, with the most important goals first, and select a point that achieves the first goal. This approach can select the next play fast. But it often selects the small but local urgent plays when there are bigger plays elsewhere.

- Multi-goals-priority

  Programs let all point generators suggest points with goal-priority value and select the point which has the highest cumulative value. This approach is slower than the first one, but it often select the point which can satisfy several purposes.

- Selective Search

  Programs choose some candidate points by multi-goals-priority approach and use these points in a mini-max (or α–β) search to select the next play. This approach seems to be reasonable and work better than others, the disadvantage is that it uses much more time than others.

2.4 PATTERN MATCHING

A **pattern (joseki)** is a classic opening play sequence in one corner of the board.

As an example we take, three patterns shown in Figure 2.8. The numbers labeling the stones express the order in which the stones are put on the board.

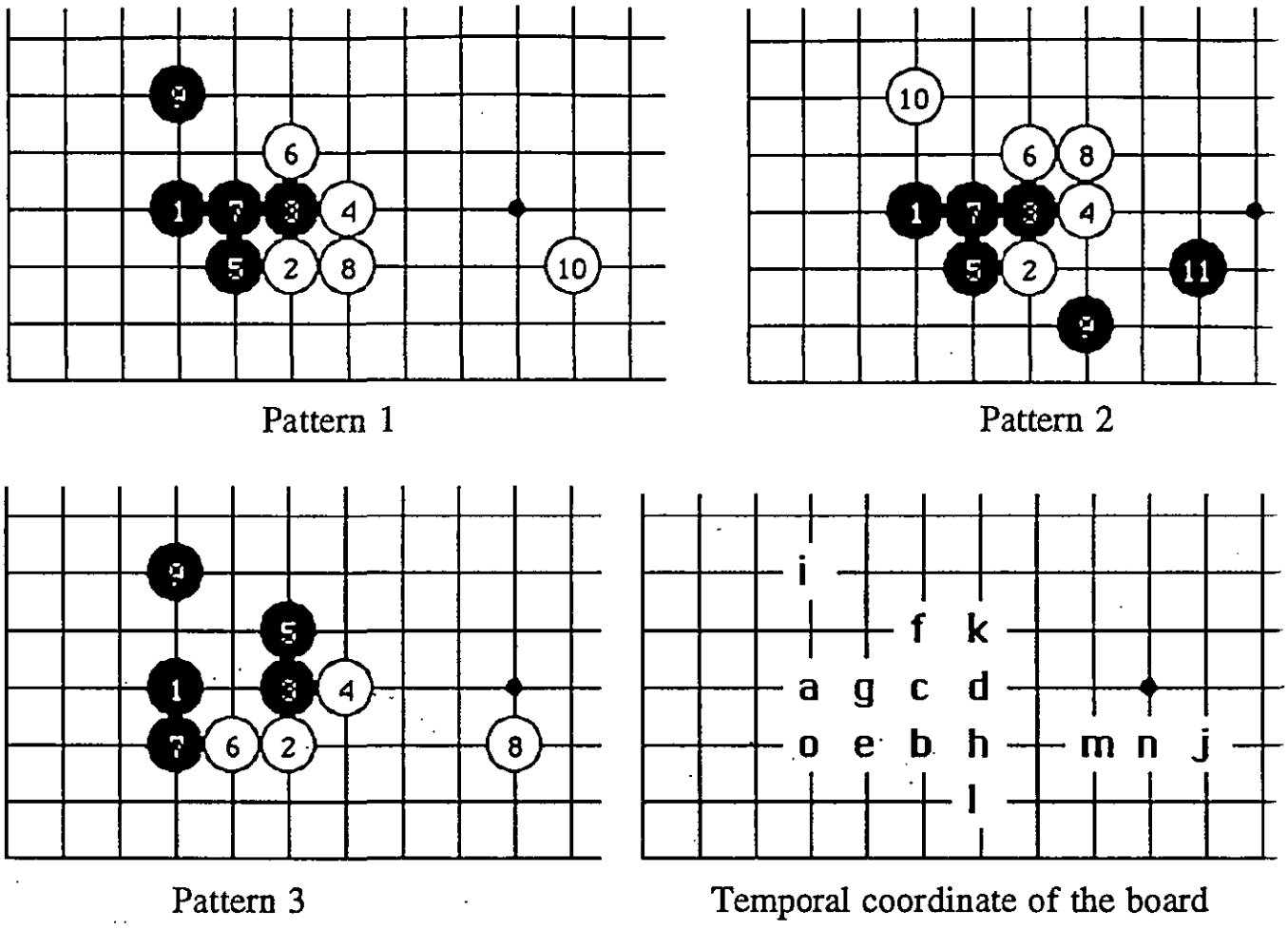
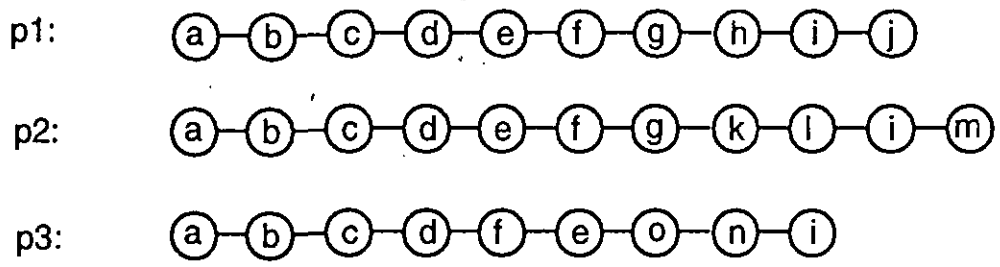


Figure 2.8: Three sample patterns

If board points are labeled as in the last picture, then the three patterns can be represented as follows:



Merging these into the following tree

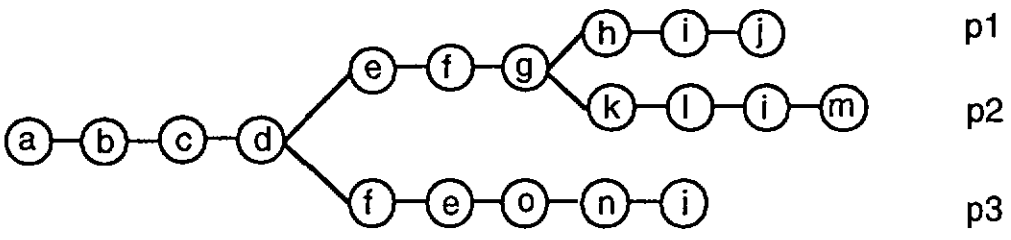


Figure 2.9: Three single patterns are merged as a multi-pattern tree

A pattern library is a knowledge library which the program can use to suggest the next play. A pattern library always contains the knowledge (patterns) and the results for each pattern. Pattern Matching is the operation to compare the present situation with the library patterns, and gives a result if the present situation "matches" a library pattern.

The pattern library in Go program is a multi-pattern tree. There are two methods to do pattern matching in the Go program.
The first one **PCP** (Playing Continuous Pattern) is a way that the program has a pointer to point out the current node in the tree and move it each time when a stone is added at the corner (we assume that only pattern plays are made). For example, in Figure 2.1, Black begins to play a stone at point *a*, then the pointer (just call it *p*) points to *a*; White finds *p* (the pointer to *a*) which has a successor *b*, then White will play at point *b* and the program lets *p* point to *b*; ... etc. If the program finds pointer *p* has several successors, then it will choose a successor point randomly among them. This method is easy to program and efficient, but it cannot handle positions where stones are not played consecutively.

The second one **PDP** (Playing Discontinuous Pattern) is a way that the program locates the current node in the tree by checking the stones' positions of the corner at every turn. This method can deal with any pattern at any moment (without killing). Its defects are that it is difficult to program, less efficient than the first method and incapable of matching the pattern if "killing" occurs.

2.5 INFLUENCE SYSTEM (IS)

Go is a strategy game. For a strategy game, the concept of "influence" is important and each player should understand it. An influence system characterizes the influence of each stone on the board.

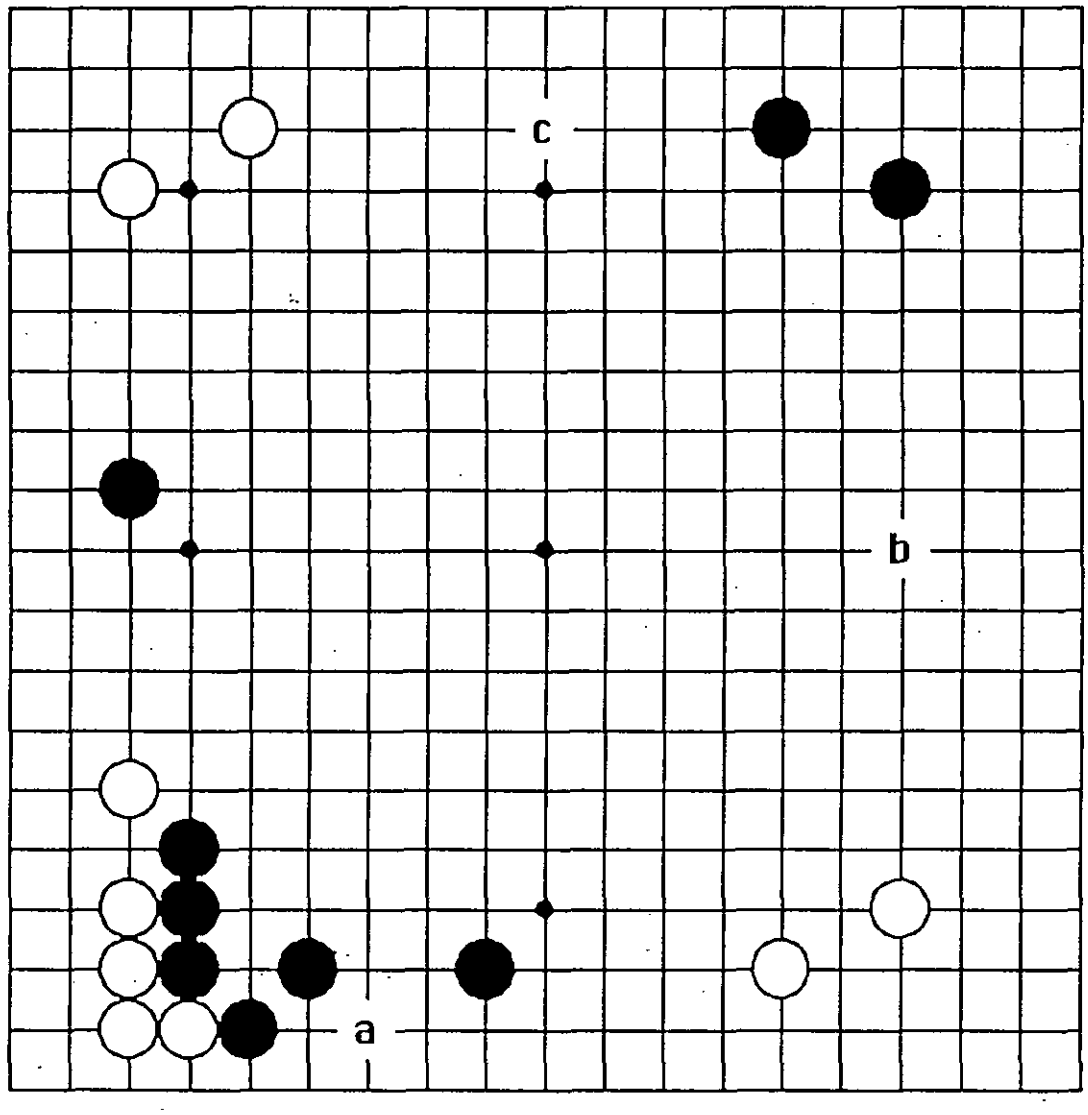


Figure 2.10: A simple example to understand the influence.

We first define

> A *x*-**nearby** point of a given point *x* is an empty point which is the *x* point itself or one of its adjacent points.

For Figure 2.10, every player knows that the a-nearby points are bad points for both sides, because they are under strong Black influence. White, playing at an a-nearby point will not have enough space to make two eyes and will be in danger;

Black playing at an a-nearby point will be considerated as "ineffective", because it does not occupy enough territory. The good points for occupying territory are the points which are under faint influence from both sides. In Figure 2.10, b-nearby points and c-nearby points are good points for both sides. Another use of influence is to calculate the territory for both players. For example, in Figure 2.10, a-nearby points are under strong Black influence, so they are considered as Black's territory. Influence is a such an important concept in Go that the early Go programs tried to form the influence concept into an Influence System (IS).

The first IS was implemented in the first Go program by Zobrist in 1969 [Zobrist 69]. Zobrist's idea is: Each black stone is given +50, each white stone is given -50, and all empty points are given 0. Then each positive (resp. negative) location adds 1 (resp. -1) to each of its four adjacent point. The 1-point add/subtract process is repeated three more times. His idea is interesting but not efficient and it is difficult to understand.

Jon Ryder improved Zobrist's idea in 1971 [Ryder 71]. Figure 2.11 shows the influence, exerted by a single isolated stone played at 60, upon nearby points.

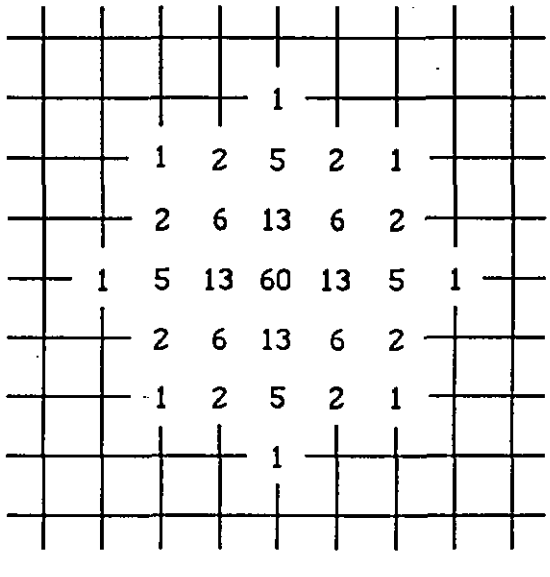


Figure 2.11: Ryder's influence system

Ryder's IS was faster and easier to understand than Zobrist's.

The last improvement was made by K. Chen in 1989 [Chen 89]. Chen's idea is to define a path from a stone to a space, and the influence of the stone on this space is in inverse exponential proportion to the length of this path. To calculate the influence of a stone, one should first calculate the "distance" between this stone and a space (empty point). I quote K. Chen's "Manhattan distance" definition and "influence" definition [Chen 89]:

"Let p1, p2 be two board points. We define the effective Manhattan distance d(p1,p2) between p1 and p2, to be the length of shortest path between p1 and p2 without going through any grid point occupied by a stone. The influence of a stone at st on a space, i.e. empty grid point at sp is given by the following formula:

influence(st, sp) = m * $f^{d(st, sp) - 1}$ ".

Here m is the influence amplitude and f is the amortization factor (its absolute value is less than 1). K. Chen chooses $m = 64$ and $f = 1/2$, Black adds positive numbers, White adds negative ones. An example of the calculation distance is given as follows:

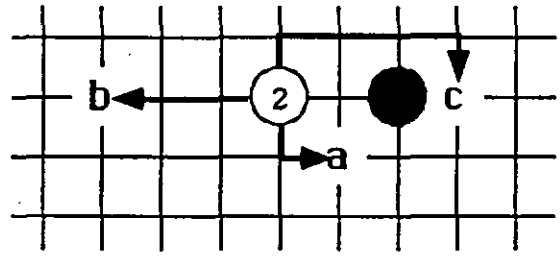


Figure 2.12: An example of distance between two points

The distances between (2) and spaces "a", "b", "c" are 2, 3, 5. The influence of stone (2) on spaces "a", "b", "c" are 32, 16 and 4.

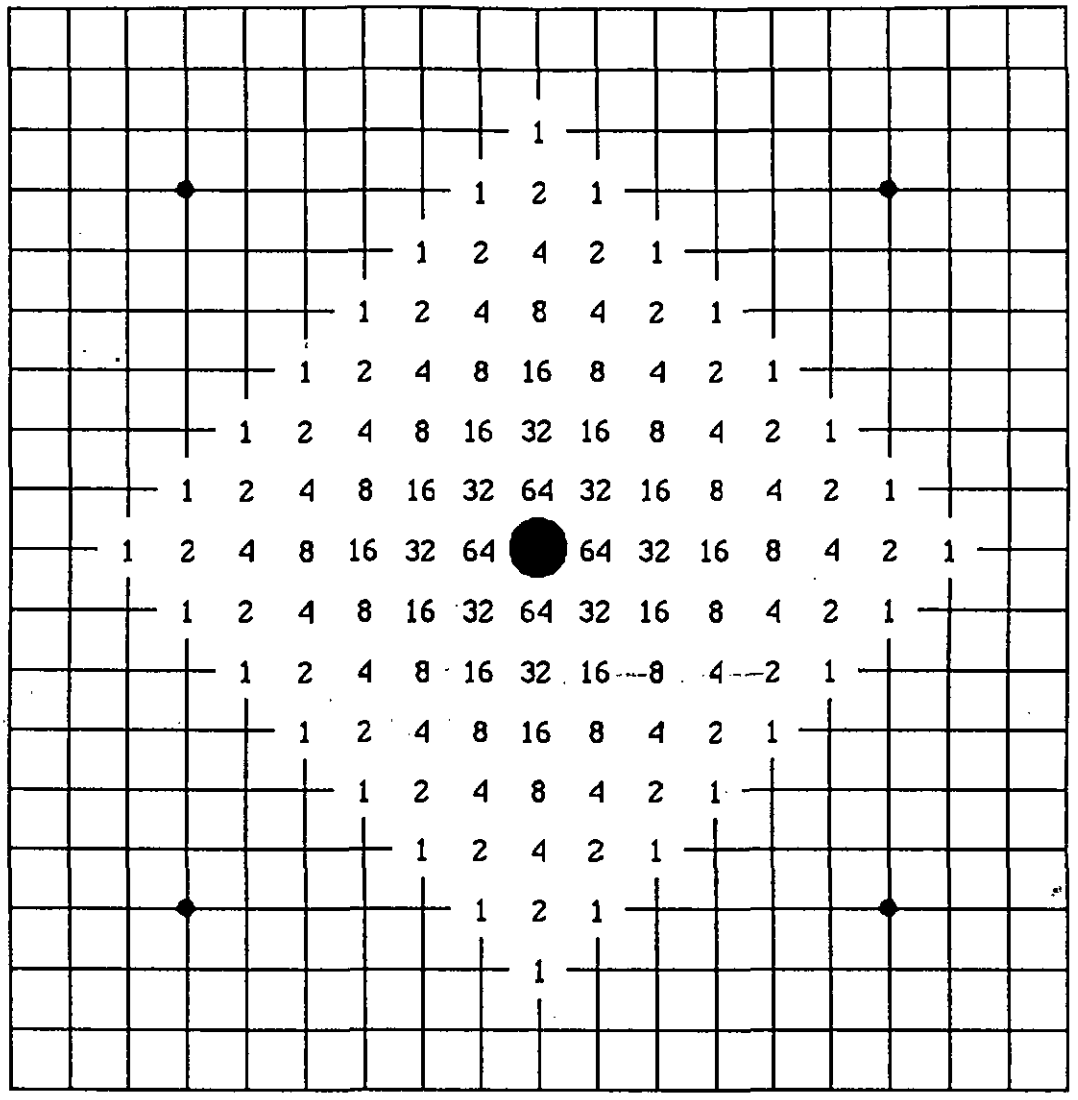


Figure 2.13: K. Chen's influence system

This IS is natural, simple, pretty and easy to understand.

2.6 GROUP IDENTIFICATION

Group identification is a tricky problem of Computer Go. To find a good and cheap group identification procedure is one of the goals of each Go programmer. K. Chen proposed a group identification procedure in 1989 [Chen 89]. His idea was to use influence to identify the group. I quote K. Chen's group definition [Chen 89]:

> "We say that two stones of same color are *a*-connected if and only if there is a path between the two stones on which each point is either an empty point of normalized influence >= *a* (resp. <= -*a*) if the two stones are black (resp. white) or occupied by a same color non-dead stone or dead opposite color stone. Note that if *a*1 > *a*2, *a*1-connectness implies *a*2-connectness. When *a* reaches some threshold, two *a*-connected stones are naturally belonging in the some group. Experience shows that n/4 is an appropriate threshold. Go Explorer uses n = 64 with any net influence value 198 or above normalized to 64, -198 or below normalized to -64, in between values are mapped into the interval (-64, 64)."

Figure 2.14 shows groups identified by influences in his program.

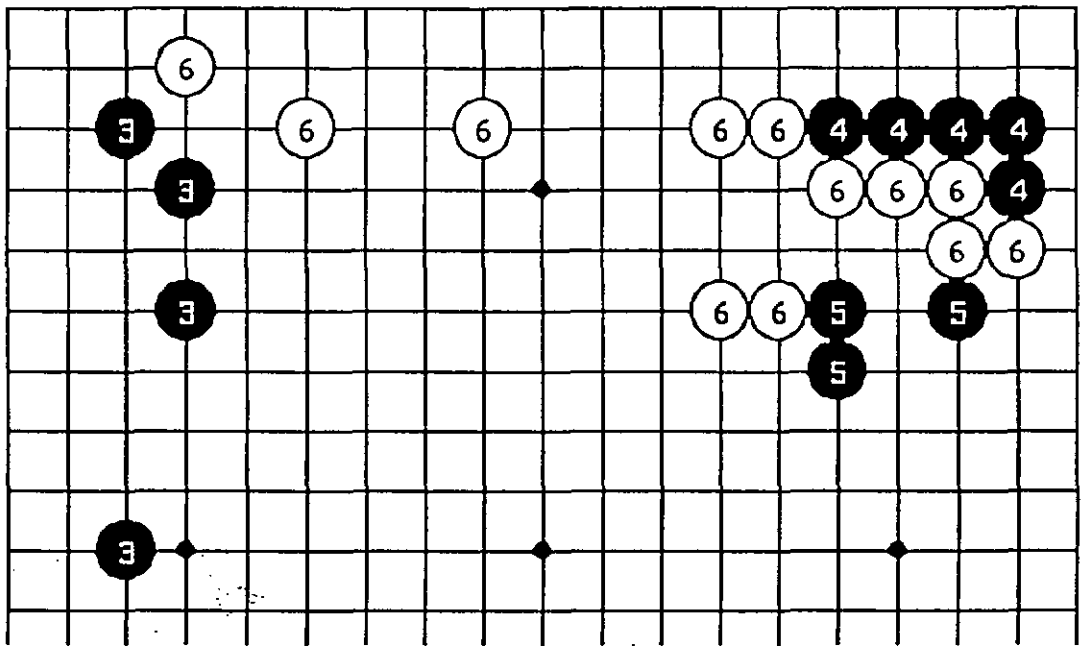


Figure 2.14: Group identified by K. Chen's group identification procedure (the numbers labeled in stones express the group number of the stone)

This group identification procedure is very practical, and can identify groups like a good human player. In section 3.3, another relationship based on a Power System is introduced.

2.7 LIFE AND DEATH (LD)

The early Go programs did not know how to deal with Life and Death (LD) problems; but now all good programs have a LD process for resolving the life and death problems, even if they are still premature.

A look-ahead process is used to resolve the LD problem. Firstly, the program will get candidate points from the heuristic generators and estimate these points in a mini-max search or an α–β search with a limited depth or a limited time. The points generated by the heuristic generators are often useful, for example, to make eyes, or to extend the space inside the group.

The LD problem makes the difference between the human and the computer. If the computer can one day find the solution to the LD problem as quickly as the human expert, then the computer Go program will be able to challenge human experts.

2.8 CAPTURING STONES IN A LADDER

"Ladder" is a special phase in Go. One player tries to capture an opponent block which has two (or three) liberties, and the other tries to escape this group. It can be defined as follows:

> A **Ladder** is a repeated process : one player wants to capture an opponent block with two (or three) liberties by playing a stone that reduces the liberties of this block to one (or two). The other player wants to save this block by playing a stone to increase its liberties from one (or two) to two (or three).

According to this definition, the maximum liberty of the escape block may be two or three. I call the former a "**two-liberty ladder**" and the latter a "**three-liberty ladder**".

Figure 2.15 and Figure 2.16 show an example of a two-liberty ladder. In this section, the numbers labeled in the stones signify the order in which the stones are placed.

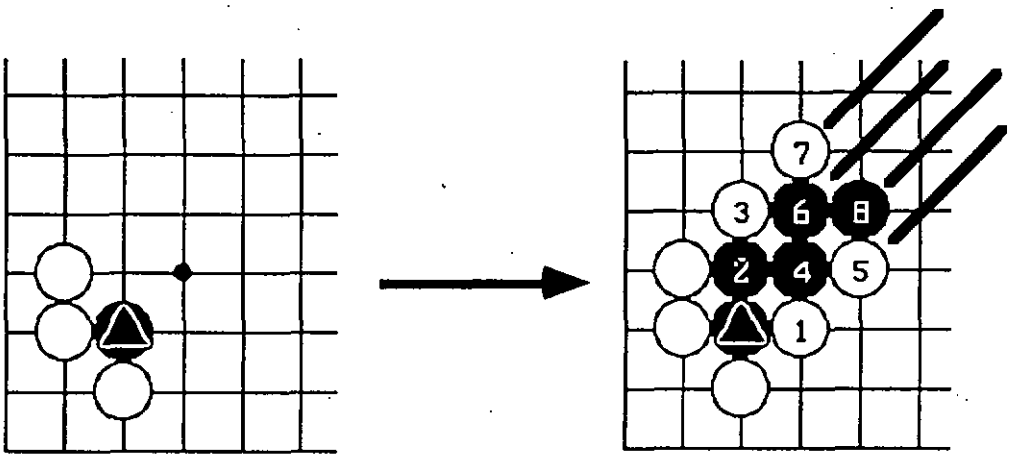


Figure 2.15: White to play and capture ▲. *Figure 2.16: A sample of a ladder*

To capture ▲ in Figure 2.15, white has to play at point 1 of Figure 2.16. If white plays at point 2 by mistake, black can escape from the ladder by playing at point 1.

The **Ladder** definition offers a method to capture a block in the ladder. Solving ladders is easy - solving ladders efficient is the problem. An efficient ladder algorithm was given in [Kierulf 90].

3. PROGRAM EQUILIBRATOR

3.1 INTRODUCTION

Equilibrator is a computer Go program which can play Go games with another player. I name it "Equilibrator" because I think that Go is a game which demands equilibrium, even if the program does not understand the concept "equilibrium" at present. The implementations of program is shown in chapter 4.

In particular it introduces a customized influence system named Power System(PS). The corresponding function, a result of a physical analogy will not only be used to evaluate a given board configuration, but also to determine group membership and to escape attacks from the adversary. Power System itself will be explained in section 3.2, its application to group membership in section 3.3 and to exit from attacks in section 3.4. Section 3.5 specifies the implemented pattern recognition and section 3.6 implements choices for the life and death look-ahead process.

My idea was to create a program which can understand the concept of Go and has a global view on the board. This idea led me to write Equilibrator during these past years. My program seems to work in this way now. Equilibrator has a global view on the board and plays game strategically rather than tactically. As Equilibrator has not fight-technique pattern and the depth of look-ahead process is limited by the speed of computer, it plays sometimes incorrectly for local fights. The strengths and the weaknesses of Equilibrator will be described in chapter 5.

The following is the general algorithm of Equilibrator after opponent plays a stone:

1. identify the group of the played stone.
2. radiate the power of the played stone.
3. identify all groups' status.
4. do pattern matching; if the program can get a play from pattern library, then, take this play as computer play; else
   * if there are weak groups, then, call life and death process and get a attack-defence play and its value.
   * do full-board look-ahead to get a global play and its value.
   * compare the value of attack-defence play and the value of global play and take the play which has higher value as computer play.
5. repeat steps 1, 2, 3 for the played stone.

3.2 A NEW POWER SYSTEM (PS)

3.2.1 Definition

Power system is the new name given to the influence system (see section 2.5). Both the words "power" and "influence" mean to have an effect on something. I choose the word "power", because it means that the effect exists physically and can be described mathematically while the "influence" cannot. For simplicity I will now refer to "influence systems" as "power systems".

3.2.2 Analogy With the Real World

From the last section, we know that a Go program must include a power system to be an effective program. To understand the role of power in the Go world, let us look first at the state of the real world. In the real world, every object has an energy and a magnetic or a gravitation field. It means that every object in the world radiates a sort of power to its surroundings. For example, a bulb spreads its light in all directions, when the light hits an obstacle in its path, there are two possible results: light changes direction (when it hits a mirror) or is absorbed by the obstacle (a piece of black cloth). The intensity of light from a bulb which touches an object is in inverse proportion to the distance between the object and the bulb. I call it *light-effect*. The same is true for Go. Go is a complex system and a miniature of the real world. We can imagine that each stone of Go is a power source similar to a bulb.

Go follows the light-effect in its system. A stone is a bulb. When a stone is put in the Go-ban, it radiates its power to its surroundings. See Figure 3.1.

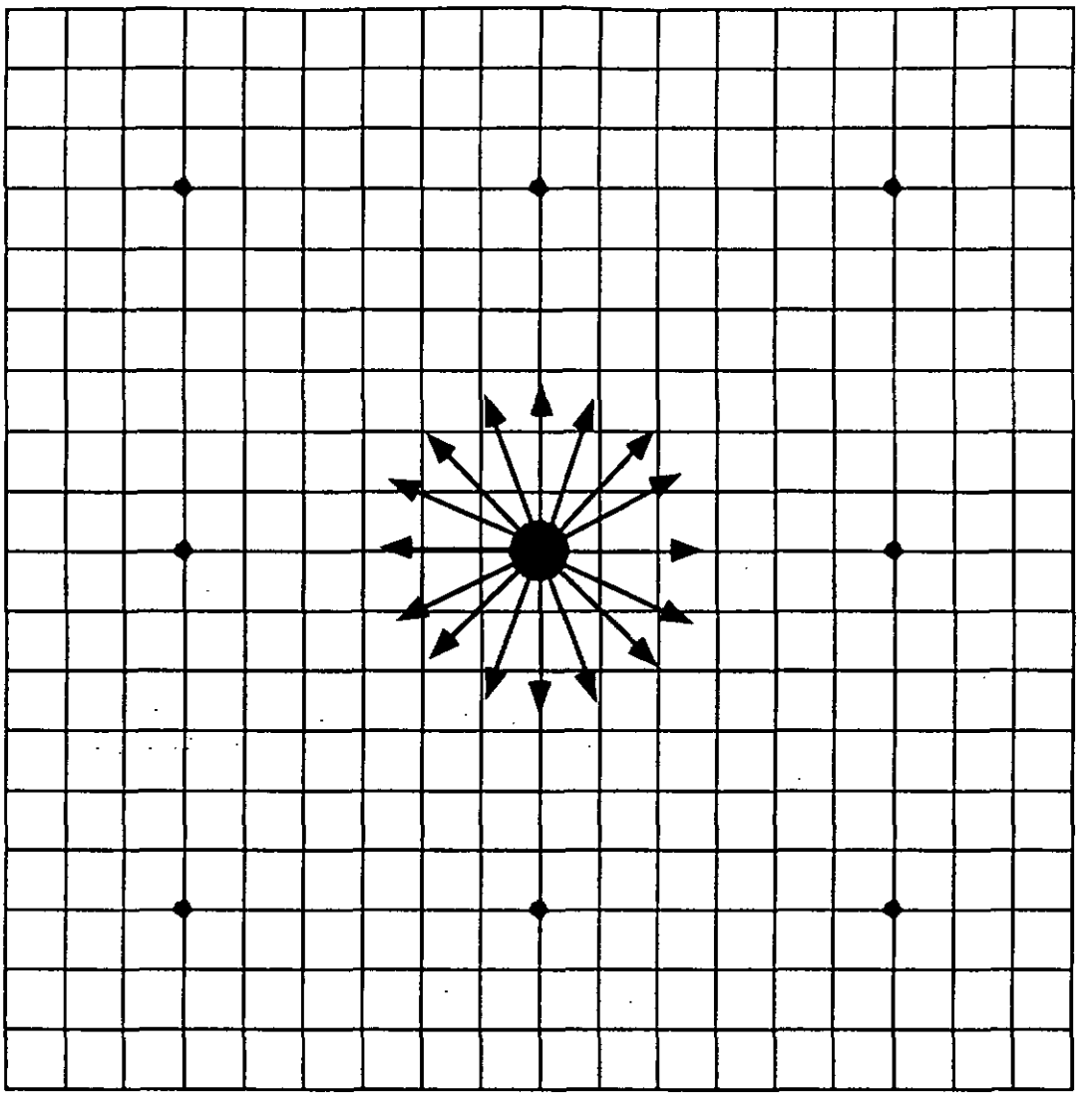


Figure 3.1: The radiant power of a stone

3.2.3 Computation of Power

To calculate the radiant power of a stone in a space, we have to find the distance between the stone and the space. If we consider that the distance from one point to its adjacent point is 1, we can calculate all distances from one point to another with the Pythagoras's formula.

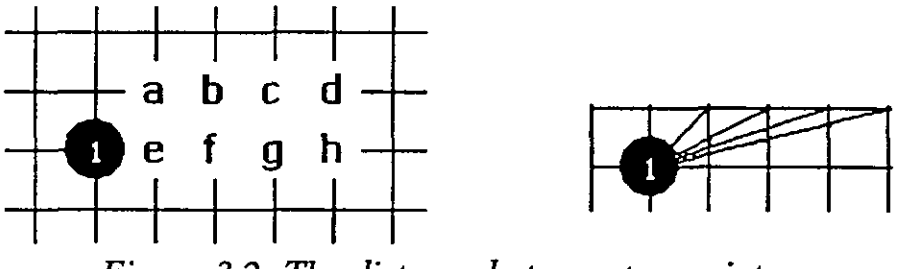


Figure 3.2: The distance between two points

If we consider Figure 3.2, we have for example: $d(st1, f) = 2$, $d(st1, c) = \sqrt{10}$.

Now, we calculate the power of stone 1 in Figure 3.2 over the other points. Using K. Chen's formula ($power(st, sp) = m * f^{d(st, sp) - 1}$) and choosing m = 100 and f = 0.5, the following results are calculated:

power(stone1, e) = $100 * 0.5^0 = 100$;
power(stone1, f) = $100 * 0.5^1 = 50$;
power(stone1, g) = $100 * 0.5^2 = 25$;
power(stone1, h) = $100 * 0.5^3 = 12.5$;
and
power(stone1, a) = $100 * 0.5^{1.414 - 1} \cong 75$;
power(stone1, b) $\cong 43$;
power(stone1, c) $\cong 22$;
power(stone1, d) $\cong 11$;

This is simplified to:

power(stone1, a) = 80;
power(stone1, b) = 40;
power(stone1, c) = 20;
power(stone1, d) = 10;

Figure 3.3 shows the radiant power of a stone.

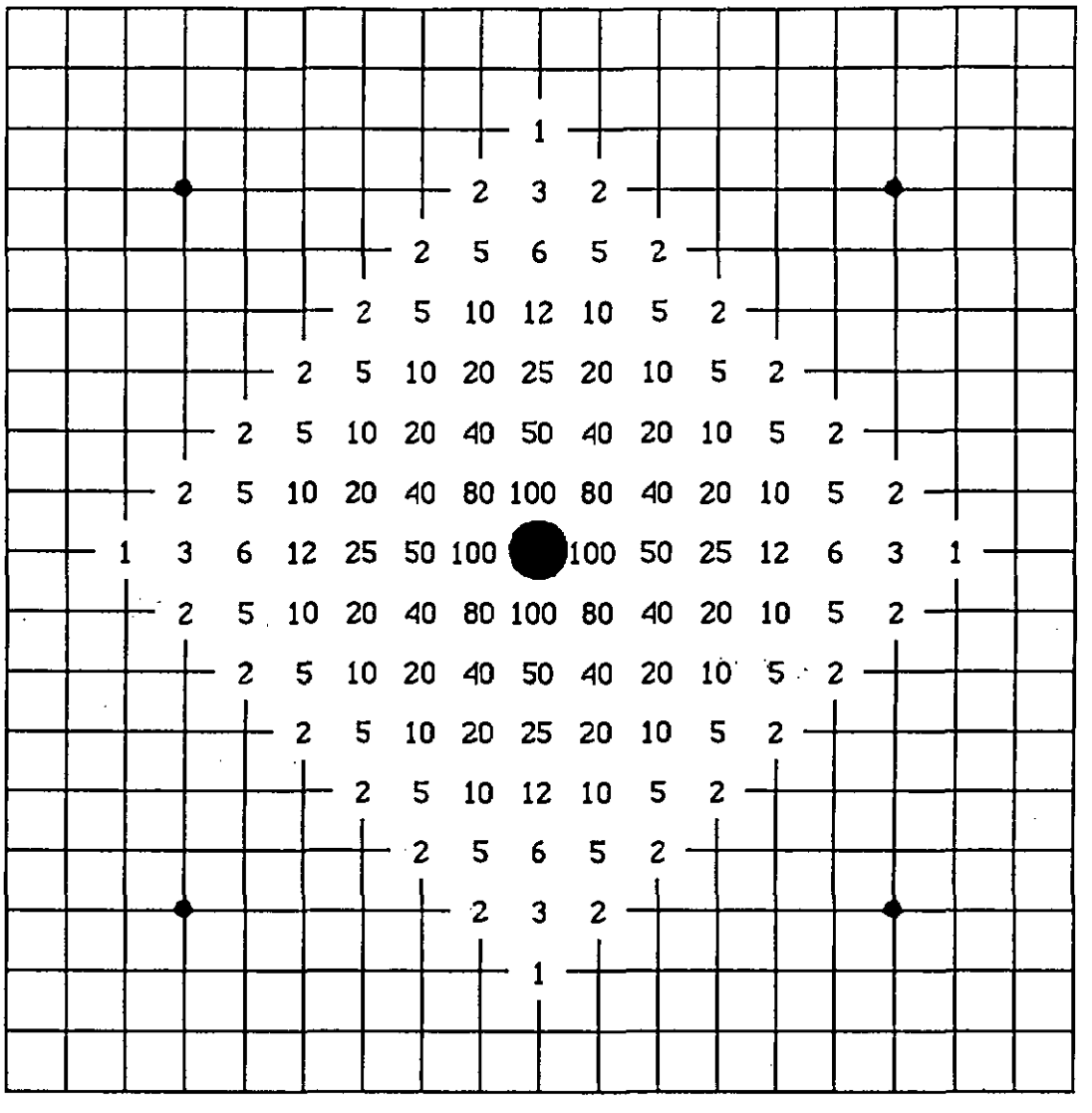


Figure 3.3: The radiant power of a stone

3.2.4 Power as Relationship

Remember now this diagram from Figure 3.2:

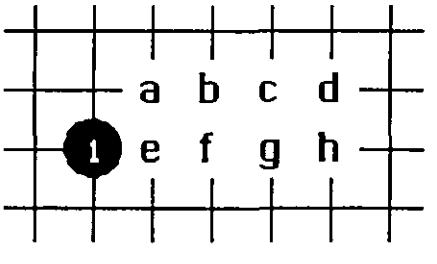


Figure 3.4: The relationship between two points

Every Go player knows the following inequalities (here "»" means "to be stronger"):

relationship(stone1, e) » relationship(stone1, a) » relationship(stone1, f) » relationship(stone1, b) » relationship(stone1, g) » relationship(stone1, c).

This example shows the limit of K. Chen's system for presenting Figure 3.3.

For example in Figure 3.4, d(st1, *a*) = d(st1, *f*) = 2, therefore point *a* and point *f* have the same power value over stone 1. It is clear that it is impossible to achieve the power system presented in Figure 3.3 by K. Chen's system. In the next section, a new power system will be introduced, which is more natural and more flexible than the above power system.

3.2.5 Design of New Power System

Using the metaphor described in section 3.2.2, it can be said that the stones of Go radiate their energy through the paths on the Go-ban.

We define the following Laws for our PS (LPS):

(1) Each stone radiates its power in 4 directions by 4 paths. These 4 powers are called **DP** (Direct Power) and at the outset they have the same value **P**. In Figure 3.5, the 4 thick lines represent the 4 DPs of the stone in the middle of the diagram.

(2) When a DP arrives in a space (empty point), it produces two further powers. These two powers are called **IP** (Indirect Power). The directions of these two IP cross in the direction of their DP. The value of IP is a fraction of DP, and a reduced rate due to changing direction is called **CR.** IP = DP * CR. In Figure 3.5, the thin lines represent IP. (be aware: IP does not produce other powers).

(3) Each power (DP or IP) advances in its direction and it reduces in value while advancing. It stops advancing when its absolute value is less than a given value called **MinLimit**. The advanced reduced rate is called **AR**.

(4) When the radiant power of a stone touches another stone (any color), the power is absorbed by that stone and stops advancing.

(5) When the DP of a stone touches the border of the Go-ban, it changes direction and advances with a fraction of itself in the reverse direction. This reflection reduced rate is called **RR**.

(6) The black stone radiates the positive power and the white stone radiates the negative power.

Figure 3.5 illustrates the mechanism of the radiant power of a stone.

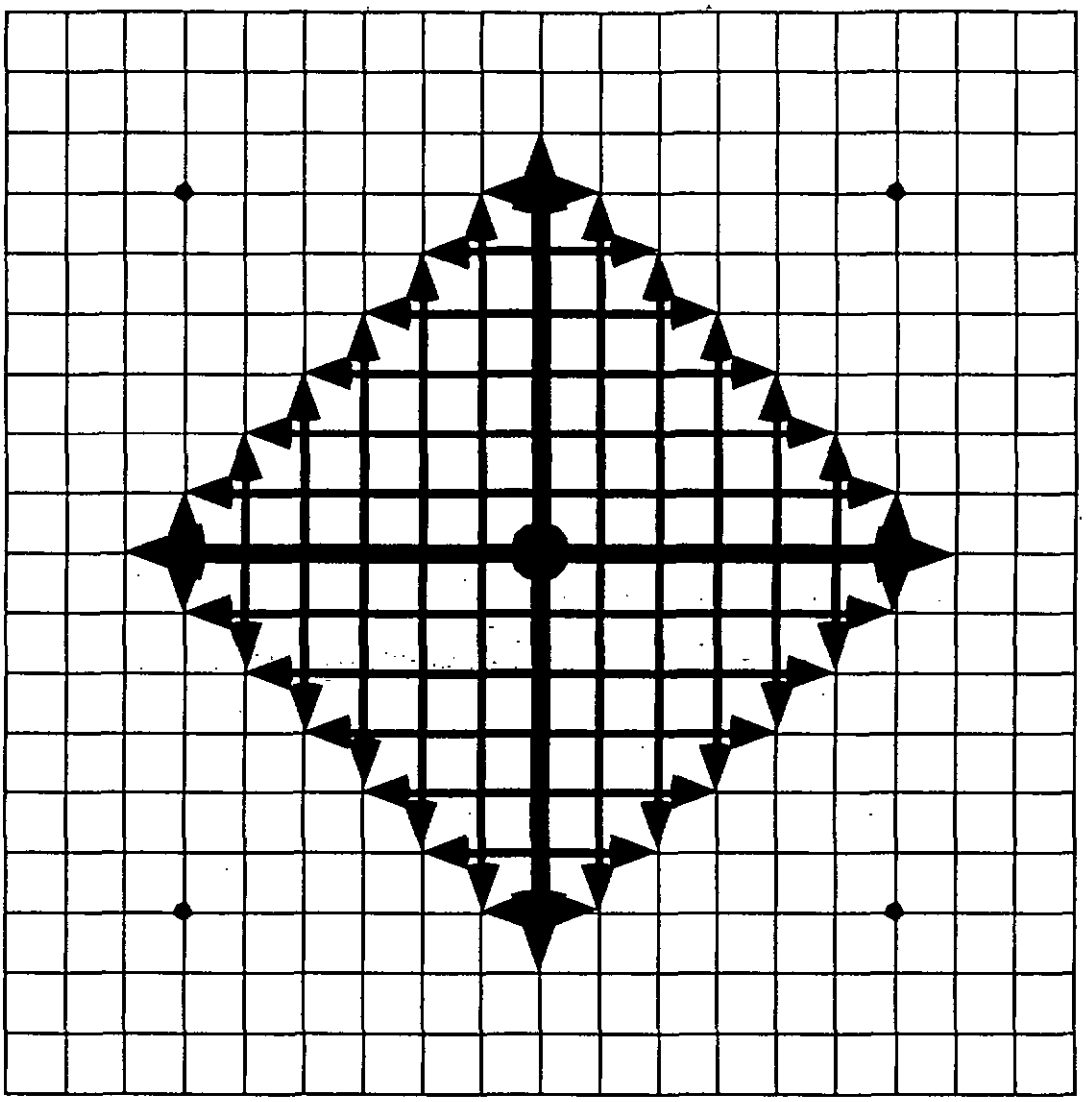


Figure 3.5: The mechanism of the radiant power of a stone

Let P =100, AR = 0.5 and CR = 0.4 in LPS, then Figure 3.6 shows the mechanism of PS.

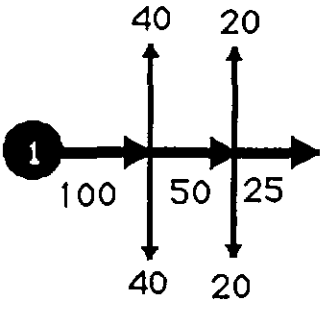


Figure 3.6: Mechanism of PS

3.2.6 A New Power Function

This is a formula which corresponds to the LPS for calculating the power of a stone at point *s* to each point *p* on the board:

$$Power(s, p) = TPN(s, p) * P * AR^{PL(s, p) - IPP(s, p) - 1} * CR^{IPP(s, p)} \quad (3.1)$$

We define:

> A **power-path** between a stone and a space is a path in which the power of the stone propagates depending on LPS (1) and LPS (2) laws defined above.

TPN (Total Path Number) is the number of power-path between two points.
TPN = 0, 1 or 2.
PL (Path Length) is the length of the power-path between two points.
PL = 1, 2, 3, ...
IPP (Indirect Power Pass):

$$IPP(s, p) = \begin{cases} 1 & \text{if IP of stone s can arrive at point p;} \\ 0 & \text{if not;} \end{cases}$$

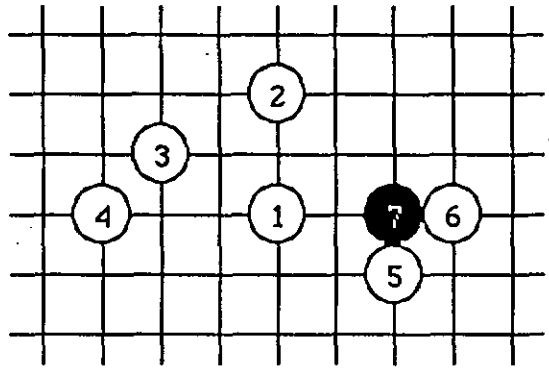


Figure 3.7: A sample of calculation power

In Figure 3.7, the different value of every function is as follows:

TPN(s1, s2) = 1;	PL(s1, s2) = 2;	IPP(s1, s2) = 0;
TPN(s1, s3) = 2;	PL(s1, s3) = 3;	IPP(s1, s3) = 1;
TPN(s1, s4) = 1;	PL(s1, s4) = 3;	IPP(s1, s4) = 0;
TPN(s1, s5) = 1;	PL(s1, s5) = 3;	IPP(s1, s5) = 1;
TPN(s1, s6) = 0;	PL(s1, s6) = ∞;	IPP(s1, s6) = 0;

The power between two points can be easily calculated using the formula (3.1):

Power(s1, s2) = 50;
Power(s1, s3) = 40;
Power(s1, s4) = 25;
Power(s1, s5) = 20;
Power(s1, s6) = 0.

Note that TPN, PL, IPP and Power are symmetric in their arguments and therefore:

TPN(x, y) = TPN(y, x);
PL(x, y) = PL(y, x);
IPP(x, y) = IPP(y, x);
Power(x, y)= Power(y, x).

As Black stones and white stones radiate different sign power, three other important formulas are:

$$\text{WhitePowerAt(point)} = \sum_{\text{i=White stone}} \text{power(point,i)} \qquad (3.2)$$

$$\text{BlackPowerAt(point)} = \sum_{\text{i=Black stone}} \text{power(point,i)} \qquad (3.3)$$

$$\text{TotalPowerAt(point)} = \sum_{\text{i=stone}} \text{power(point,i)} \qquad (3.4)$$

Here, "point" means every point on the board, empty or not.

3.2.7 Power System Algorithm

Let DPoint be coordinate of DP transporter, IPoint be coordinate of IP transporter. When a stone is put at point *s* on the board, it will radiate its power as following procedure:

```
PROCEDURE Power(s : Position; PieceColor : Color; GroupNb : INTEGER);
BEGIN
      FOR DD:= up, down, left, right DO
         DPoint := s;
         advance DPoint one point (grid) along direction DD;
         IF DPoint is not out of the board THEN
            DPower := DP (or -DP, according to PieceColor);
            REPEAT
               put DPower with GroupNb at DPoint;
               IF DPoint is edge point of the board THEN
                  DPower := DPower * RR;
                  DD := reverse direction of DD;
               ELSE
                  DPower := DPower * AR;
               END;
               IF DPoint is not occupied THEN
                  FOR ID :=   Turn clockwise 90° of direction DD,
                              Turn anticlockwise 90° of direction DD DO
                     IPoint := DPoint;
                     advance IPoint one point (grid) along direction ID;
                     IF IPoint is not out of the board THEN
                        IPower := DPower * CR / AR;
                        REPEAT
                           put IPower with GroupNb at IPoint;
                           IPower := IPower * AR;
                           advance IPoint one point (grid) along direction ID;
                        UNTIL | IPower | < MinLimit OR IPoint is occupied OR
                        IPoint is out of the board;
                     END;
                  END;
                  advance DPoint one point along direction DD;
               END;
            UNTIL | DPower | < MinLimit OR DPoint is occupied;
         END;
      END;
END Power;
```

When a stone is placed on the board, the program will call this procedure to radiate its power to its surrounding.

3.2.8 The Strengths and Weaknesses of the Power System

Strength

- **Flexibility**

  The first strength of the Power System (PS) is its flexibility. In PS, the power of the stone is distinguished between direct power and indirect power, so it is more subtle than others.

  If P = 100, AR = 0.5 and CR = 0.4, we calculate the power radiated by the stone showed in Figure 3.8.

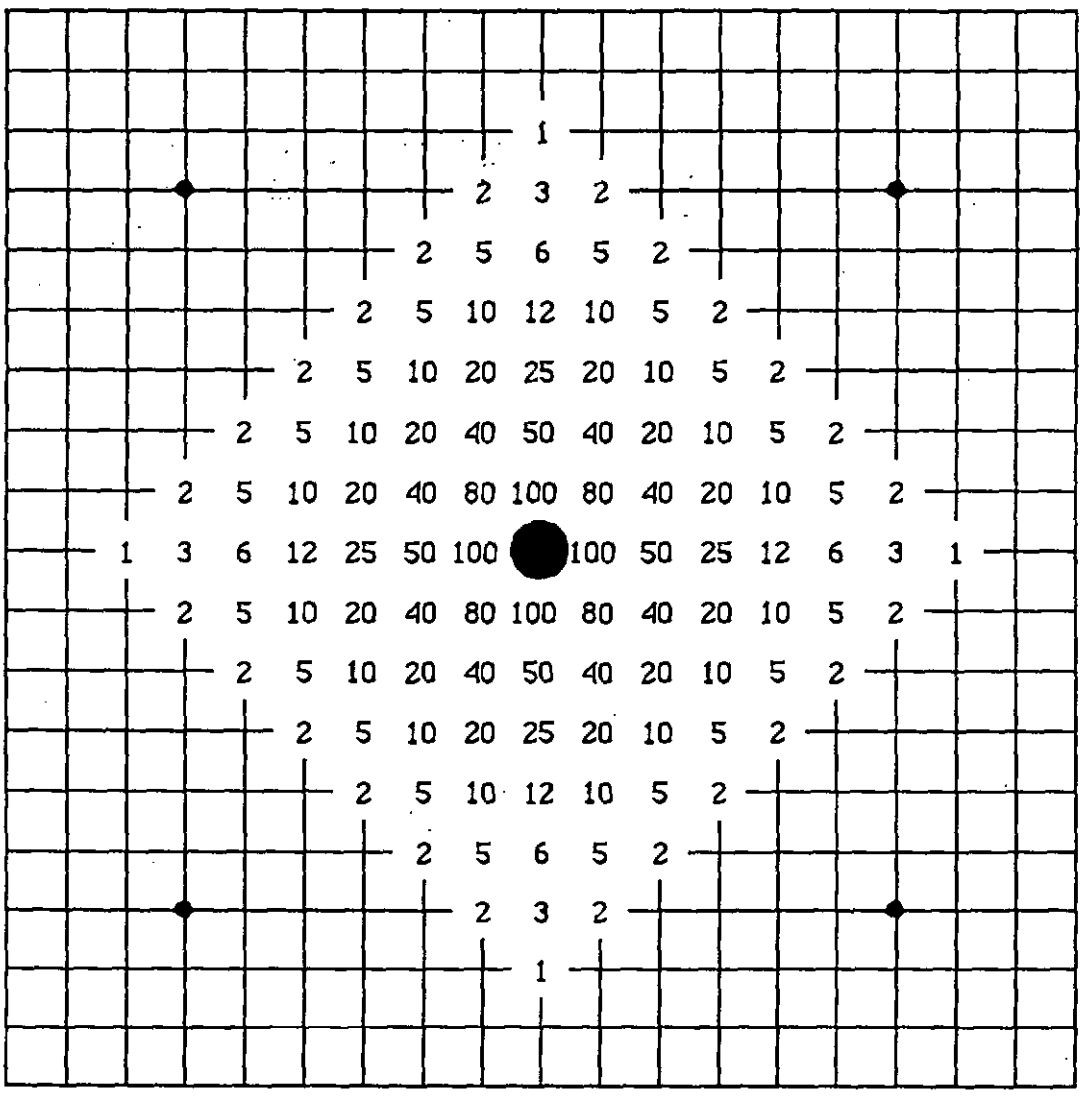


Figure 3.8: The radiant power of a stone

If P = 64, AR = 0.5 and CR = 0.25, we create the same "picture" as that created by K. Chen's power system (see Figure 3.9):

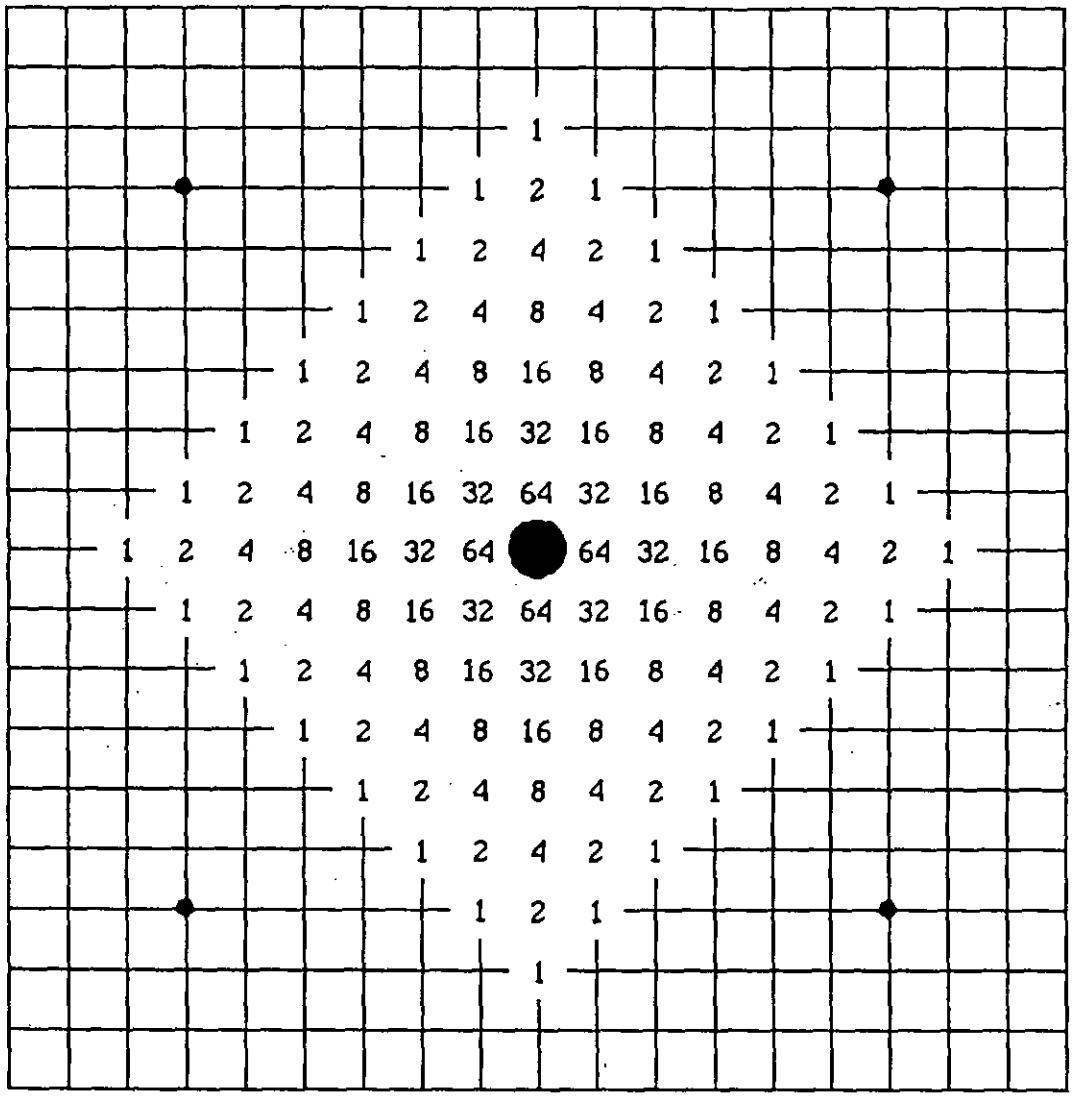


Figure 3.9: K. Chen's power system

In spite of the similarity of two pictures, the underlying systems are different.

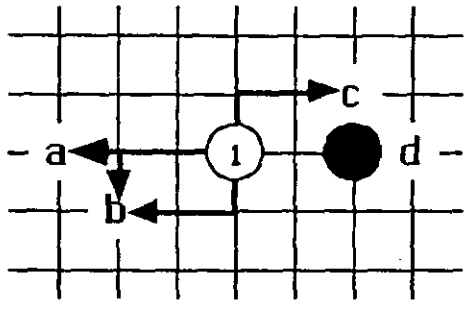


Figure 3.10: Mechanism of PS

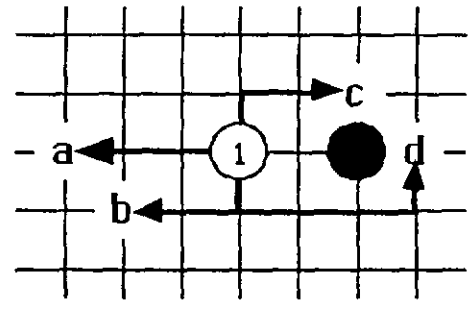


Figure 3.11: Mechanism of K. Chen's system

This difference is demonstrated in Figure 3.10 and Figure 3.11, where the arrows indicate the paths followed by the power. In K. Chen's system, the points *a, b, c* have the same power value of stone 1 and there is a path from stone 1 to point *d*. In PS, however the power values of stone 1 on points *a* and *b* can be the same (see Figure 3.9) or different (see Figure 3.8). The power values of stone 1 on point *c* are the half of the value of those on point *b* and there is no power-path from stone 1 to point *d*.

- **Power Direction**

The second strength of the Power System (PS) is the fact that its power has the direction. This power direction allows the program to check from which direction the powers are coming to the point. It is also useful for finding the exit of the group. (see section 3.4).

- **A Fast Group Identification Procedure**

PS allows the program to use a fast method to identify the group, as seen in section 3.3.

Weakness

The first weakness of PS is that it sometimes calculates the power twice on a point instead of once as with K. Chen's system. For example, in Figure 3.10, there are two paths from stone 1 to point b in PS, so Equilibrator has to calculate twice the power value on point b. There is only one path from stone 1 to point b in K. Chen's system, so K. Chen's program calculates only once. The second weakness is that PS uses more memory than other power systems. The third weakness is that it is difficult to program.

3.3 GROUP IDENTIFICATION

Remember the definition of group given in section 2.2.1

*Two stones of same color having a "strong relationship" belong to the same **group***

where the basic concept is the key word "relationship". Another membership concept using PS as defined in section 3.2 is now presented:

"A stone belongs to a group if the total power of the same color group over the stone surpasses a given value."

This value can be set to 40 (P * CR). For example, in Figure 3.12, the power of stone △ over stone □ is 50, therefore the stone △ and stone □ belong to the same group. On the other hand, in Figure 3.13, the power of stone △ over stone □ is 25, so they are not in the same group.

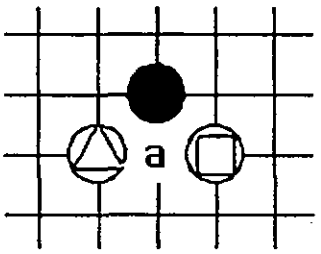


Figure 3.12: stone △ and stone □ belongs to the same group

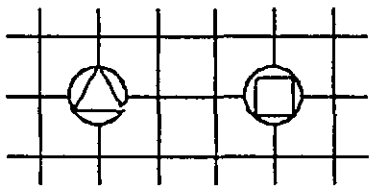


Figure 3.13: stone △ and stone □ do not belongs to the same group

The following Figure 3.14 shows the groups identified by Equilibrator.

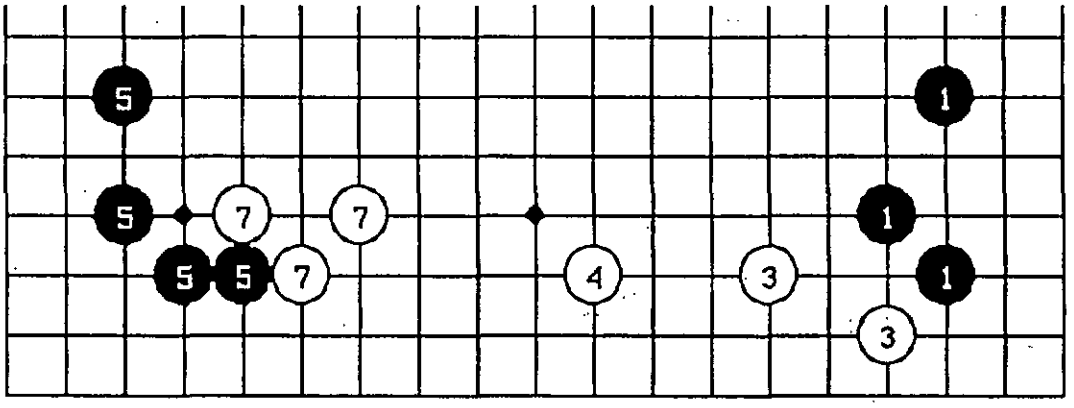


Figure 3.14: Groups identified by Equilibrator

This group identification differs from that given in section 2.6. The advantage of this method is that the program does not need to find paths and not to calculate the power on these paths; it is therefore faster. Nevertheless, while it can recognize the member of the group, it does not detect the danger coming from outside, Figure 3.12 is an example of this. For the other group identification procedures, the stone △ and stone □ do not belong to the same group, because in the "path point" *a*, the total powers are not strong enough for White. But in Equilibrator, they are considered as members of same group. As shown in Figure 3.15 (I), the relationship between the two white stones will be broken if Black plays at point *a*. They are then considered as different groups by other group identification procedures, but as the same group by Equilibrator.

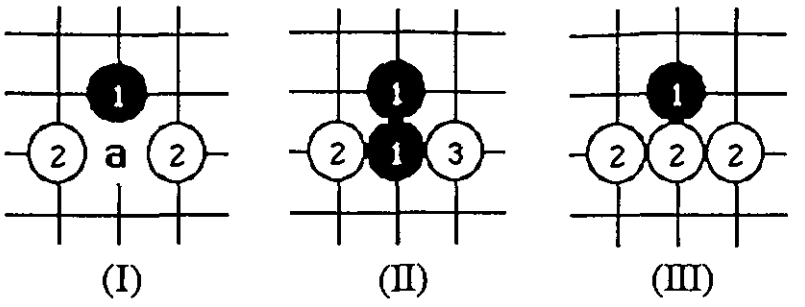


Figure 3.15: The state of group recognized by Equilibrator

There is a physical analogy behind this group identification. The relationship between the two white stones in Figure 3.15 can be translated into the physical connection shown in Figure 3.16. The first case (I) is particular in the sense that the connection between *a* and *b* is neither strong (III) nor broken (II) but in an intermediate, weak state. In that sense, the two white stones in Figure 3.15 (I) belong to the same group.

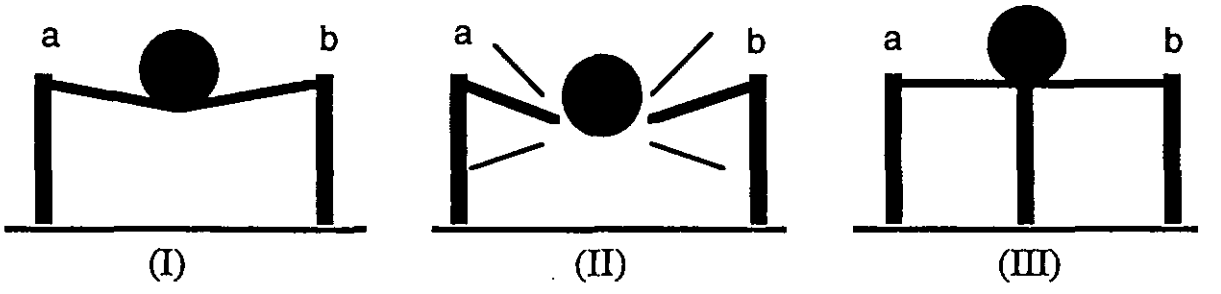


Figure 3.16: Interpretation of Figure 3.15

3.4 FINDING THE EXIT OF GROUP

When a group is in danger, it should make two eyes to live or it should find a "exit" to escape from the opponent encirclement.

A further application of PS is presented, in order to provide a method to find the "exit" of a group. As the power has direction, we can easily determine the "exit" of a group. An exit direction is the direction in which the opponent influence vanishes or has a very little volume.

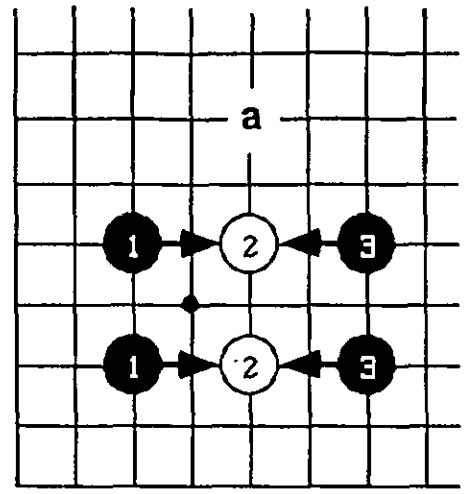


Figure 3.17: Determines the group's exit by the direction of power (the numbers labeled in the stones are group number of the stones)

For example in Figure 3.17, group 2 is almost surrounded by the opponent group and it has only one "exit" direction toward the top of the board, because there is no black power radiating from the top. To escape from Black's encirclement, White has to play a stone in an *a*-nearby point.

3.5 PATTERNS

Equilibrator uses PDP (Playing Discontinuous Pattern) to do "Pattern Matching". Here is the pattern data structure and the procedure used in Equilibrator:

```
PROCEDURE GetPieceFromPattern(LastX, LastY : CoordinateTy;
                              VAR LastPieceInPattern: PatternTy;
                              VAR FindPattern : BOOLEAN);
```

`LastX` and `LastY` are the coordinates of the opponent's last stone placed in the board. This procedure verifies the present situation in the corner where the opponent placed the last stone (piece). If the procedure does not find a pattern which matches the present situation, then the parameter `FindPattern` is set to false; if it does, then `FindPattern` is set to true and the parameter `LastPieceInPattern` points to the current node of the pattern; the program then chooses its next play randomly from the successor nodes of `LastPieceInPattern`.

A pattern library as knowledge base is very useful for computer game-playing. But how to use the pattern is a difficult problem even for professional players.

3.6 LIFE AND DEATH

As other Go programs, Equilibrator uses a heuristic algorithm to solve the LD problem. Some heuristic generators are used to determine candidate points relative to the selected weak group and to evaluate these points with an α–β search, and to determine if the group concerned is captured or alive.

For computer playing game, α–β search is an important approach. It examines a subset of the possibilities which Mini-Max search does, but it does not avoid combinatorial explosion (see procedure Alpha-Beta). Go is a complex game; unlike chess programs, Go programs cannot examine all possibilities of the board in each depth of α–β search. Instead of examining all possibilities, Go programs examine some selected possibilities. Equilibrator uses an α–β search algorithm, applied on a group of special selected points relative to the considered weak group. These points are important strategic points for both players and it is expected that all these points are played by one or another player. This α–β search determines in which order these points are to be played and it is named "Selective α–β Search". The principle of Selective α–β Search is as following: after some candidate points (relative to the selected weak group) are chosen before the beginning of search, the program selects in each level the empty points *only* from these candidate points as successor points in Selective α–β search (see procedure Selective-Alpha-Beta). As in all games, the symmetric possibility exists in Go. Figure 3.18 shows an example of two symmetric positions.

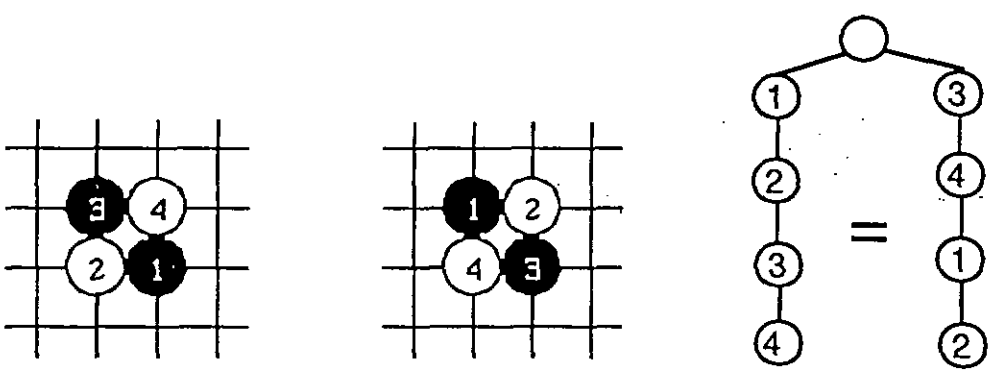


Figure 3.18: two symmetric positions represented by two branches of a game tree

In Figure 3.18, one of two branches should be cut off from game tree to avoid symmetric positions. Equilibrator uses non-symmetric positions in Selective α–β Search (I call it "Selective Non-Symmetric α–β Search") to avoid combinatorial explosion (see procedure Selective-Non-Symmetric-Alpha-Beta). Now, let's compare the difference among standard α–β search, Selective α–β Search and Selective Non-Symmetric α–β Search. The standard α–β search given in [Knuth & Moore 75] is as following:

```
PROCEDURE Alpha-Beta(p : Position; alpha,beta : INTEGER) : INTEGER;
VAR m,i,t,d : INTEGER;
BEGIN
  determine the successor positions p1,...,pd;
  IF d=0 THEN
     Alpha-Beta := f(p);
  ELSE
     m := alpha;
     FOR i := 1 TO d DO
       t := - Alpha-Beta(pi,-beta,-m);
       IF t > m THEN m := t; END;
       IF m ≥ beta THEN go to done; END;
     END;
     done: Alpha-Beta := m;
  END;
END Alpha-Beta;
```

Selective α–β Search is as follows:

```
PROCEDURE Selective-Alpha-Beta({p1,...,pd} : Position;
                               alpha,beta : INTEGER) : INTEGER;
VAR m,i,t : INTEGER;
BEGIN
  p:=pd;
  IF d = 1 THEN
     Selective-Alpha-Beta := f(p);
  ELSE
     m := alpha;
     FOR i := 1 TO d-1 DO
       t := - Selective-Alpha-Beta({p1,...,pd-1},-beta,-m);
       IF t > m THEN m := t; END;
       IF m ≥ beta THEN go to done; END;
     END;
     done: Selective-Alpha-Beta := m;
  END;
END Selective-Alpha-Beta;
```

Selective Non-Symmetric α–β Search is as follows:

```
PROCEDURE Selective-Non-Symmetric-Alpha-Beta({p1,...,pd} : Position;
                                              alpha,beta : INTEGER) : INTEGER;
VAR m,i,t : INTEGER;
BEGIN
  from among p1,...,pd, select a position "p" which is non-symmetric with all
  preceding examined positions;
  IF p = NIL THEN
     Selective-Non-Symmetric-Alpha-Beta := ∞;
  ELSIF d = 1 THEN
     Selective-Non-Symmetric-Alpha-Beta := f(p);
  ELSE
     m := alpha;
     {s1,...,sd-1} := {p1,...,pd} - p;
     FOR i := 1 TO d-1 DO
       t := - Selective-Non-Symmetric-Alpha-Beta({s1,...,sd-1},-beta,-m);
       IF t > m THEN m := t; END;
       IF m ≥ beta THEN go to done; END;
     END;
     done: Selective-Non-Symmetric-Alpha-Beta := m;
  END;
END Selective-Non-Symmetric-Alpha-Beta;
```

Here, $p_1,...,p_d$ are the empty candidate points which are chosen by the program. The numbers of possibilities examined among three search are very different. Let's take an example of a game tree which has width of 4 and depth of 4; Figure 3.19 -3.21 will show the differences among three approaches for this example in the worst case (meaning there are not deep cutoffs).

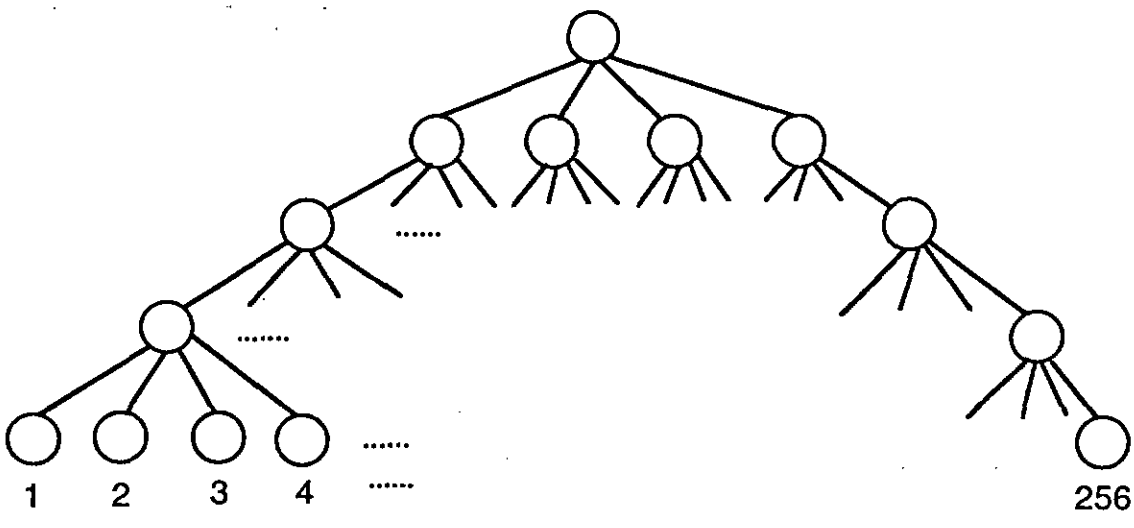


Figure 3.19: A complete game tree for α–β Search.

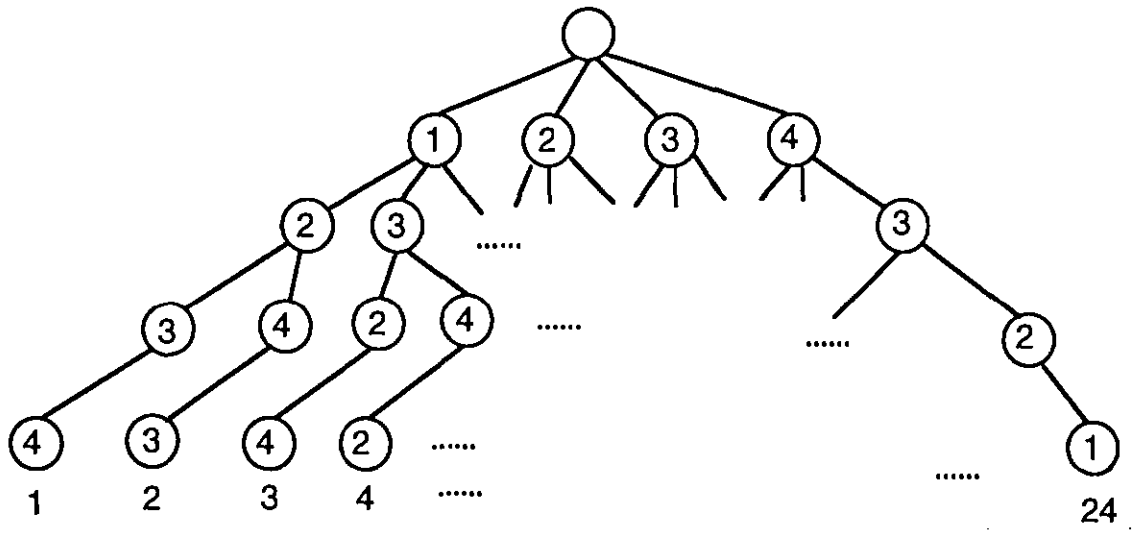


Figure 3.20: A complete game tree for Selective α–β Search.

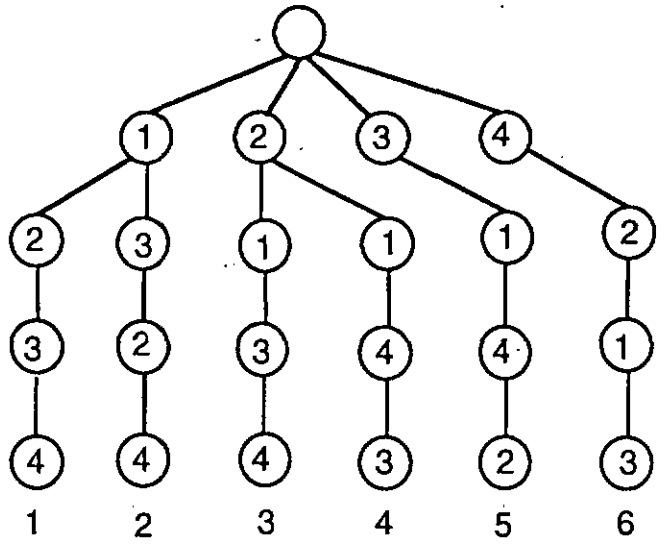


Figure 3.21: A complete game tree for Selective Non-Symmetric α–β Search.

The differences among three approaches are very great. Instead of examination 256 possibilities in α–β search, Selective Non-Symmetric α–β Search examines only 6 possibilities. This approach can avoid combinatorial explosion, but it is not flexible enough and could ignore good points. There is a relationship between the search depth and the search width. The formula of calculation all non-symmetric possibilities for d points is: $PNS(d) = \binom{d}{\lfloor d/2 \rfloor}$. This is a result of the case when no "killing" has occurred. There will be slightly more possibilities if some stones are killed during the search. The α–β search will cut off about half of the leaves of the search tree, for Selective Non-Symmetric α–β Search, the general formula is:

$$w = \frac{\binom{d}{\lfloor d/2 \rfloor}}{2} + d$$ w is search width and d is search depth.

$$w_{min} = \frac{\binom{d-1}{\lfloor (d-1)/2 \rfloor}}{2}$$ w_{min} is minimum search width and d is search depth.

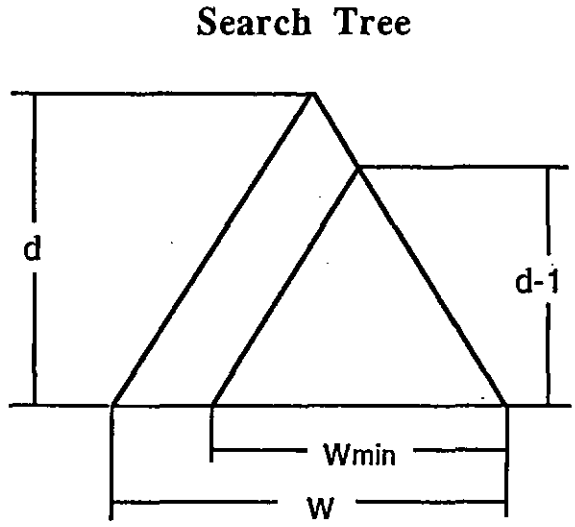


Figure 3.22: Search tree

See Figure 3.22. If we choose $w < w_{min}$, the program may possibly take the first candidate point without trying any other points in Selective Non-Symmetric α–β Search.

3.7 CAPTURING STONES IN A LADDER

Equilibrator has also a ladder algorithm like others programs. It uses the same idea than others, but it can resolve the three-liberty ladder problem. Equilibrator considers to capture not only two-liberty blocks but also three-liberty blocks. Figures 3.23 to Figures 3.28 show the examples of how Equilibrator (Black) captures △ block in a ladder.

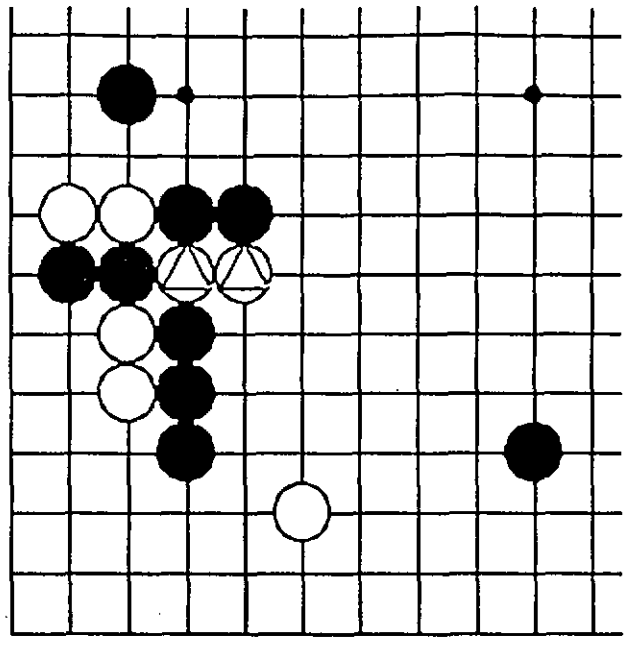


Figure 3.23: Black to play to kill △ block (Two-liberty ladder)

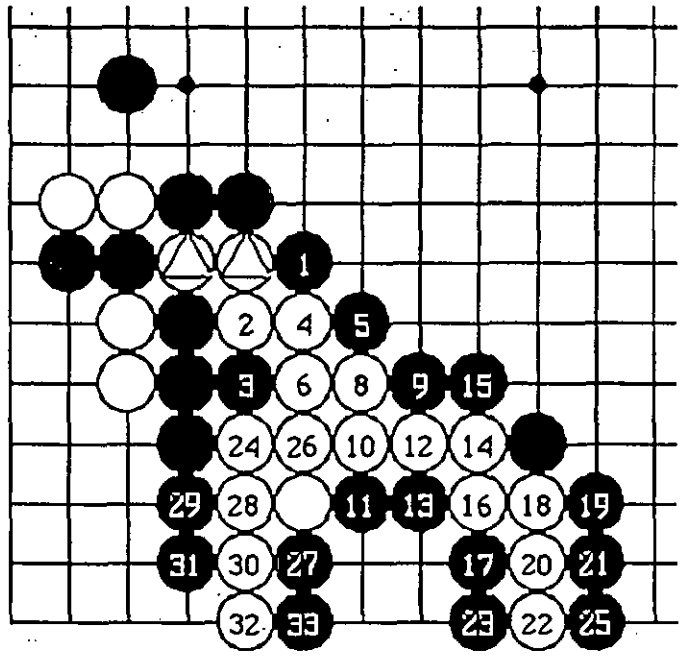


Figure 3.24: Result of Figure 3.23, 7 = 26.

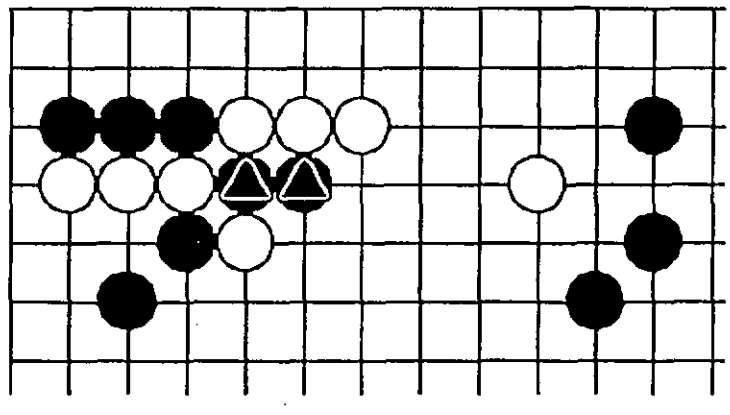


Figure 3.25: White to play to kill △ block. (Three-liberty ladder)

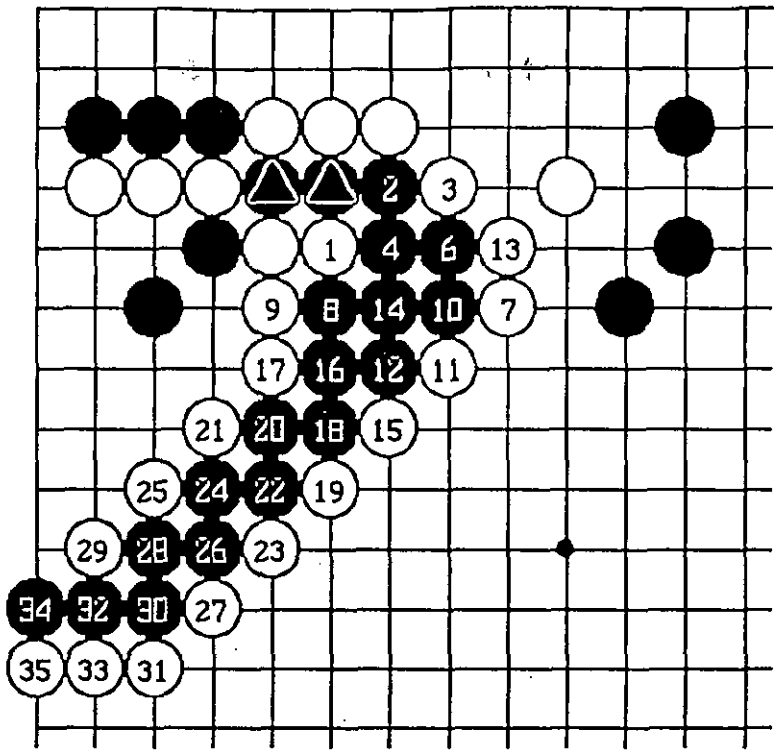


Figure 3.26: Result of Figure 3.25, (5) = (14).

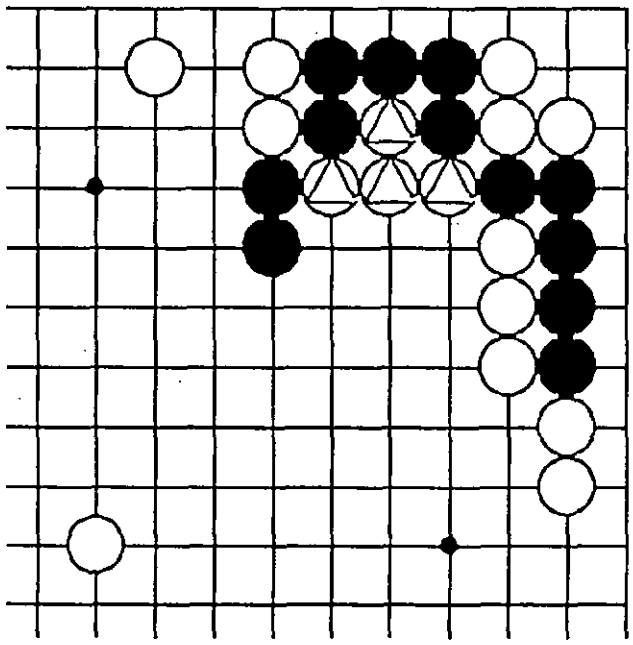


Figure 3.27: Tricky ladder. Black to play to kill △ *block*

(Three-liberty ladder)

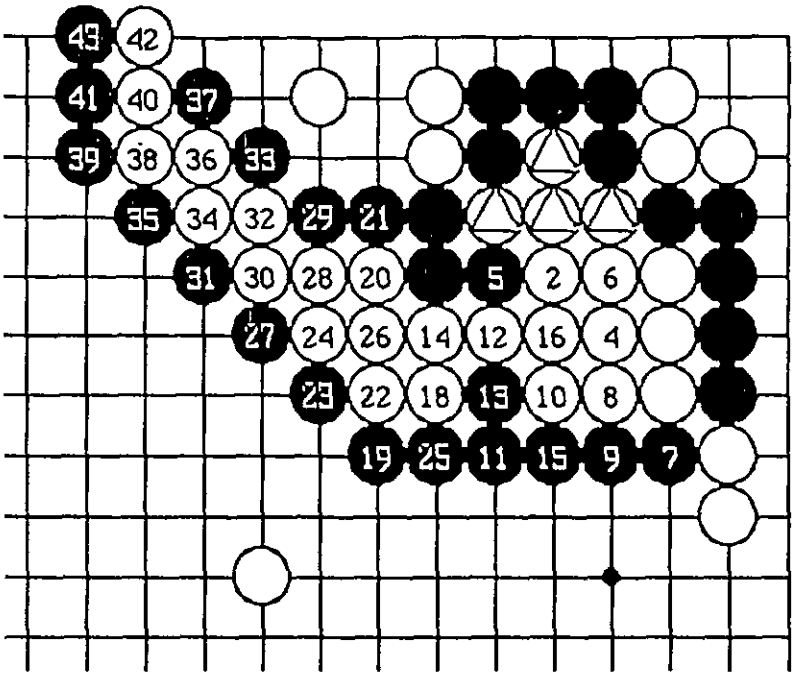


Figure 3.28: Result of Figure 3.27,
1 = (6), 3 = (16), 17 = (26).

Capturing a three-liberty block in the ladder is much more difficult than capturing a two-liberty block both for human and computer. Capturing a block in ladder is one of few aspects of Go where a computer can compete with a human expert.

4. ARCHITECTURE AND IMPLEMENTATION

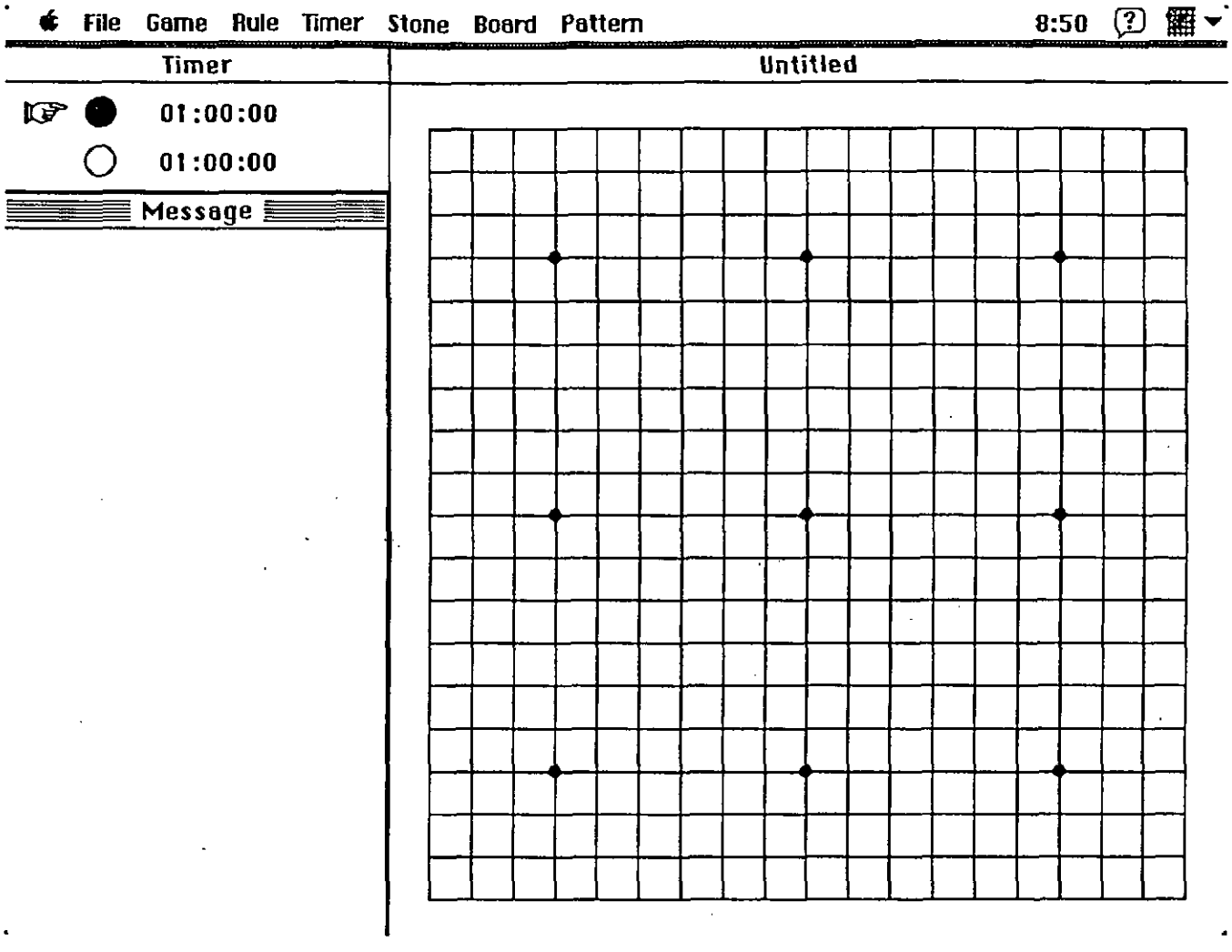


Figure 4.1: Graphic Interface

Structure of Equilibrator

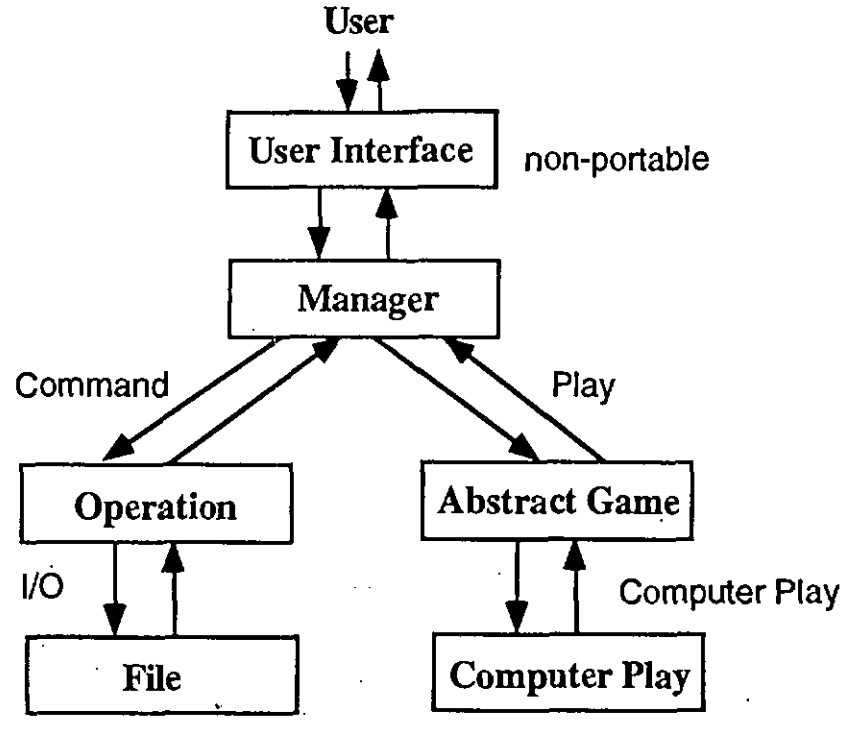


Figure 4.2 : Structure of Equilibrator

- **User Interface:**
  The user interface provides the graphic interface in which the user can communicate with the computer. It provides:
  (a) a menu to allow the user to choose a command.
  (b) a board (Go-ban) to allow the user to place a stone on the board. (click on the board).
  (c) a clock to show the available time for two players.
  (d) a message window to display the messages to the user.
  It is machine dependent and non-portable.
- **Manager:**
  It handles the user's actions. When the user clicks on the screen, there are three possibilities:
  (a) user's click is illegal (clicks in Time Window, Message Window or plays an illegal play), then Manager does nothing.
  (b) user clicks in the menu, then Manager will send a message to Operation component.
  (c) user clicks in the board, then Manager will send a message to Abstract Game component.

- **Operation:**
  If the user wants to do an I/O operation, Operation will send a message to File component, otherwise it will execute the other user's commands, for example: set the clock, display only the number of the last stone, display all stones with the order number in which the stones were placed.
- **File:**
  This component manages files. It reads a game (or the pattern library) from a file or writes a game (or the pattern library) to a file.
- **Abstract Game:**
  It manages the virtual images of the board, for example *block, group.*
- **Computer Play**
  This component implements functions supporting computer play. It gets a play from Pattern Library as its response if it finds, otherwise it uses Selective Search approach (see section 2.3.2) to select one point as its response. Figure 4.3 shows the sketch of Computer Play.

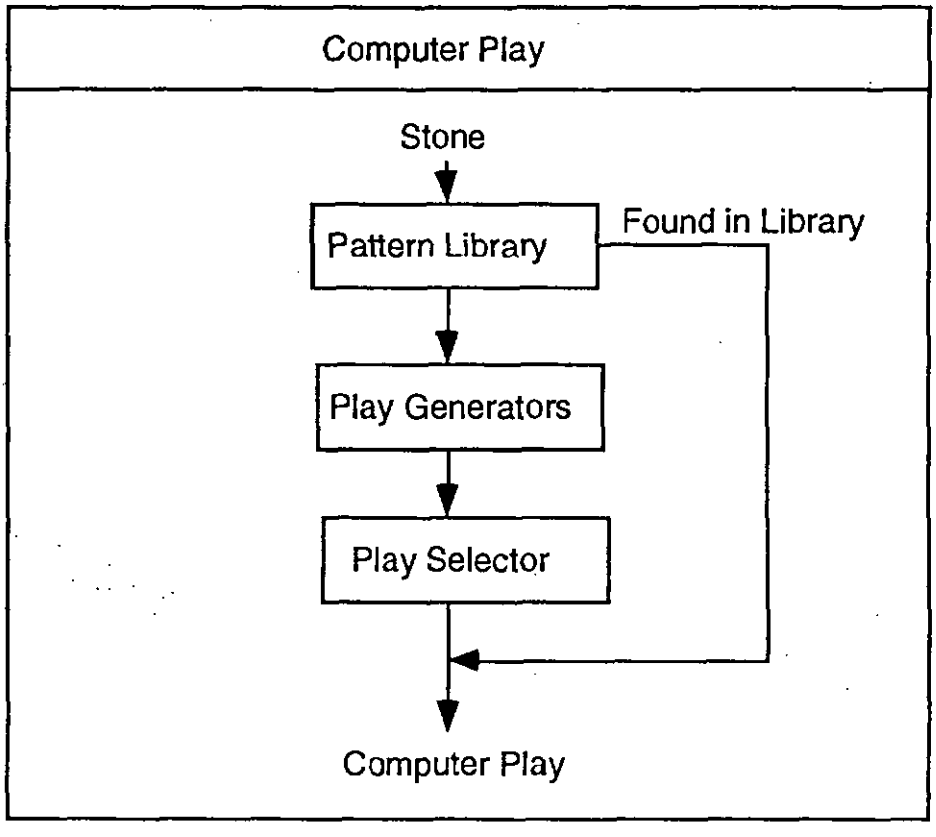


Figure 4.3: Equilibrator's Computer Play

Equilibrator is implemented in language MODULA-2 on Macintosh. MODULA was chosen as a structured language which can resolve any problem without many difficulties, and the MODULA compiler on Macintosh has a very good debugger. This is very important for developing a large complex system like Go.

The program has about 30 modules, 250 pages, 500 kBytes and 15'000 lines.

The conclusion which I draw from my experience leads me to suggest ***Hard thinking before doing.*** It is difficult to do but following this law saves time. Another experience is to choose a programming language which has a good debugger.

5. THE STRENGTHS AND WEAKNESSES OF EQUILIBRATOR

Equilibrator has a pattern library, but there are no patterns in its library for the time being; it plays with its evaluation function, its recognition of the groups' status and the look-ahead processes. It is interesting to compare the behaviour of the Equilibrator with another computer Go program which has a big pattern library. The following is a game in which Equilibrator (Black) plays with another Go program:

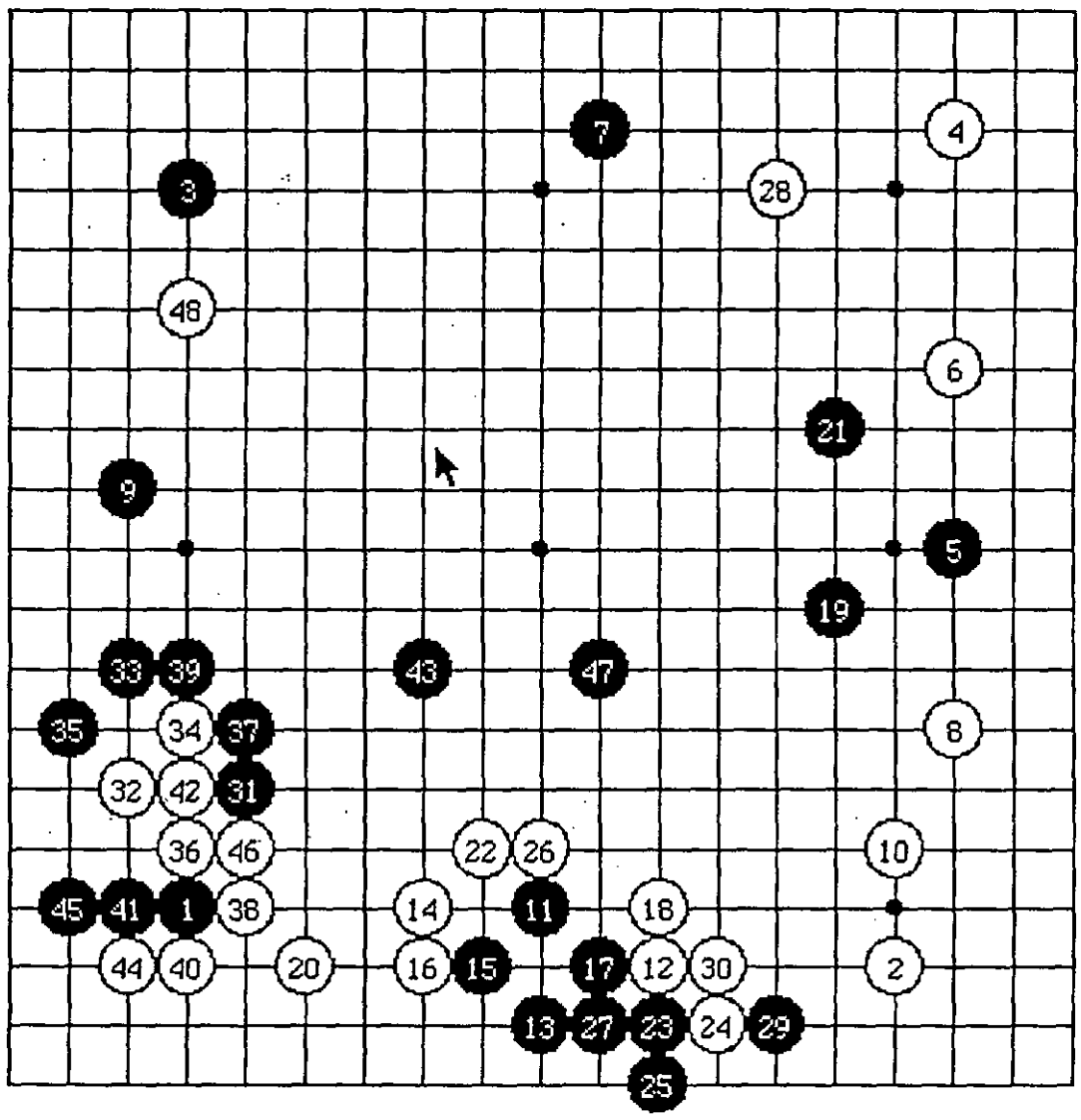


Figure 5.1: A game of Equilibrator (Black)

In the beginning of the game, Equilibrator (Black) plays quickly (occupying the strategy points) instead of following the pattern like White does. Until play 11, Equilibrator has an advantage over the opponent. After play 14, Equilibrator recognizes that there are two weak groups: stone 12 and stone 14; it begins to attack stone 14 by playing at 15, but it is not a good choice because play 15 is too near the

black group and does not exert enough pressure upon stone 14. Play 17 is as same as play 15. After play 18, Equilibrator considers that there is no weak group and plays at 19 to occupy the strategy point. After White plays at 22, Equilibrator recognizes that the lower black group has no exit and is in danger, then it plays at 23 to make an eye to live. Black's play 23 and play 25 are correct for making the eye; but play 27 is useless because the lower black group is already alive. Because of Black's plays 15, 17 and 27, White begins to catch up with Black. But from play 31 to 47, Equilibrator shows its good strategy tactics. Play 31 is a good play which has three effects: the security of stone 1, the limitation White's influence and the occupying of the territory. Play 33 and play 35 are the good attack points. After White plays at 36, Equilibrator begins to abandon the bottom left-hand corner (which is small) and controls the middle of board (which is big) by playing at 37, 39, 43 and 47. To play 47, Black retakes the advantage.

Figure 5.2 shows another game where Equilibrator (White) plays with the same program.

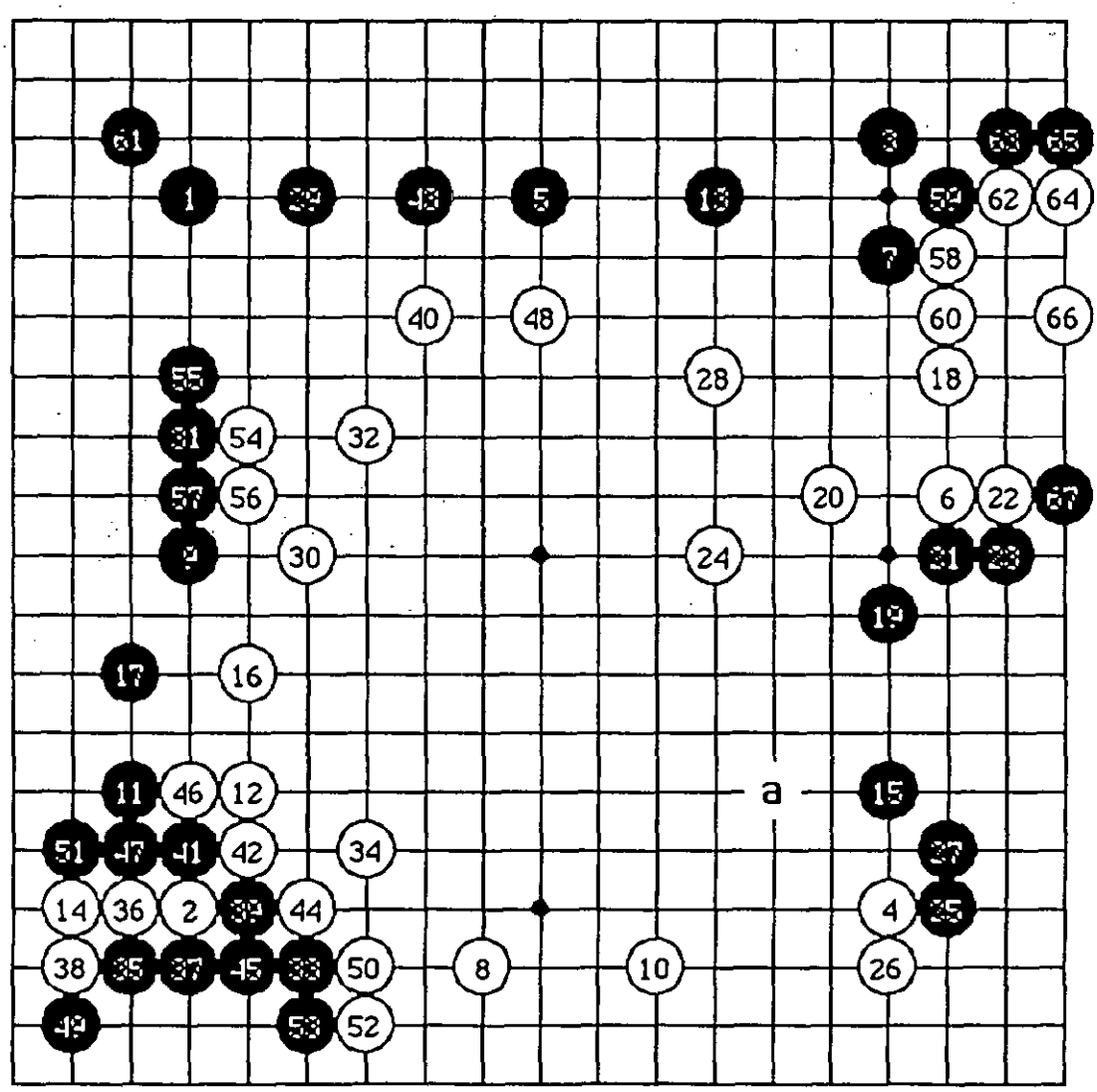


Figure 5.2: A game of Equilibrator (White)

Like first game, Equilibrator tries to play at strategy points. Until play 7, two programs play correctly. Play 8 should play at 9 and play 10 should play at 17. Play 12 and play 16 show Equilibrator's strategy. Play 18 should play near by 19 to attack stone 15. Equilibrator continues its strategy by playing at 24, 28, 30, 32. After Black plays at 33. White begins to attack it. But play 34 should play at 45 and play 38 should play at 39. After that Equilibrator continues to occupy the strategy points 40 and 48, but it loses its bottom left-hand corner which is very big. From that Black takes the advantage. After play 50, point *a* becomes very important to enclose territory for White and to destroy White territory for Black. But neither of programs plays at this very big point until play 67.

From the above discussion, we see that Equilibrator can choose good strategy points, has a good group identification procedure and can recognize the groups' status; but it sometimes makes mistakes and its fight techniques are not good. To improve its fight techniques, a fight-technique pattern seems necessary, even if a fight-technique pattern may be change the strategy of program: instand of focus on strategy points, the program focus on the points which are sometimes locally urgent but small. Every Go player knows that there are several "styles" of play in Go as: occupation of strategy points; making territory; killing opponent weak group; equilibration the game; making the game "beautiful". Some players have their styles, some players not. Equilibrator has its style of play as occupation of strategy points, even if it sometimes makes mistakes.

Writing a Go program demands a lot of time and patience (Wilcox has been working on it since 1972). I have been writing Equilibrator for three years, but three years are not enough to write a Go program which plays pretty well. Ten years would be a reasonable term to write a 1 dan program for now.

To reach a 1 dan, the programs need at least a good fast left-death detector and a good function. Write a left-death detector is not difficult, but write a fast and correct left-death detector is very hard. Finding a good function to judge correctly the situation of the board is fundamental for programs, but it seems very hard. Human players judge the game with their experience, the programmers do not know how to formalize this experience into a computer system.

6. CONCLUSION

Computer Go is a developing application of AI. We understand from what we have just discussed that Go is a very good model which allows us to simulate, analyse and resolve the problems of the real world. Once we derive some principles from the Go model, we can use these principles in different fields of AI to resolve different problems.

PS given in this thesis is important, but the mode of thinking demonstrated in the thesis is even more important. This mode of thinking is "Theory-Practice". This means before resolving the practical problems of the real world, we set up an abstract world which corresponds to the real world. However we do not worry about whether or not this abstract world can resolve the problems of the real world. We can use this mode of thinking not only to formalize Go but also to modelize further problems into a formal system and to resolve them.

Even if Mr. Ing has offered one and a half million dollars to the inventor of the first Go program able to beat a Go player of his choice before the year 2000, I do not believe that any Go program will win this prize within the allotted time. I do not even believe that any Go program could become a world champion before the year 2050. I would not make bet upon this, because it would be unfair when I am sure that I would win.

I would like to end by saying: "Do not over-estimate the capability of our intelligence product".

APPENDIX: THE INTEGRATION OF THE POWER SYSTEM INTO THE CHESS PROGRAM

The principle of computer chess was proposed by Shannon in 1950 [Shannon 50]. Since then Computer Chess has been very successful in the field of AI. Now, the best program has reached Grand Master level and it is possible that the computer wins the human world championship in the coming years. More than 30 years ago, Herbert Simon, a recognized expert in the field of AI, predicted that the world chess champion would be a computer within 10 years. But until now, his prediction has not really come true. Even if today the best Chess programs reach Grand Master level, we do not consider the Chess program can "think" like humans. They act like humans but cannot "think" like humans. What do they lack ? The "feeling". The feeling is a very important tool used by humans to make judgements which lead them to the best seeming direction. The feeling is experience. The experience is still something that we do not know how to translate into the compute science today. In computer Go, we have found a power system to interpret some human experiences and feelings. How to build a power system in chess is presented as follows:

The idea is almost the same as Go, but a little different. We consider that each chess piece radiates its power to its surroundings. We use the same vocabulary and the same principle of PS. Let P = 2. From Figure A.1 to Figure A.4 show some examples.

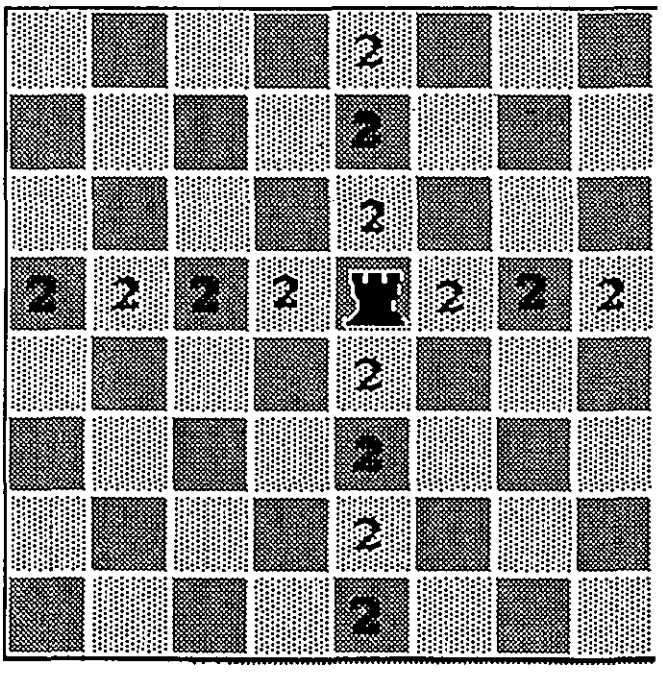


Figure A.1: The direct radiant power of black rook

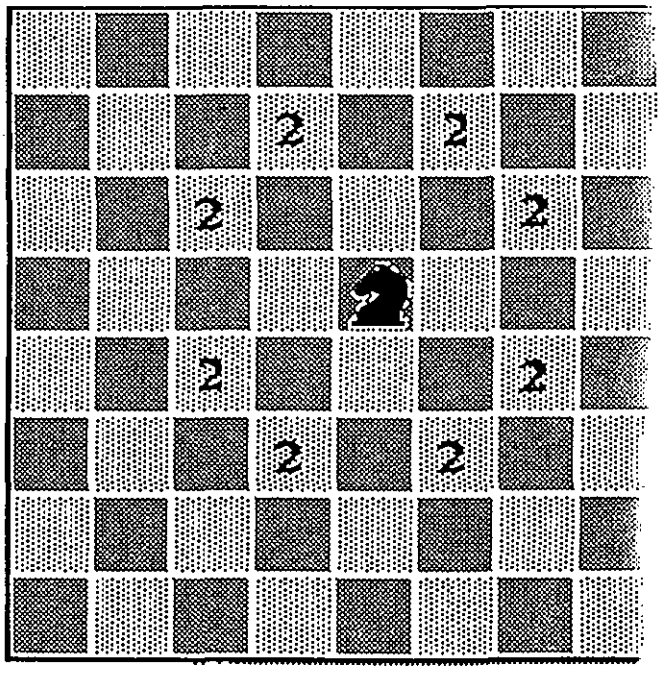


Figure A.2: The direct radiant power of black knight

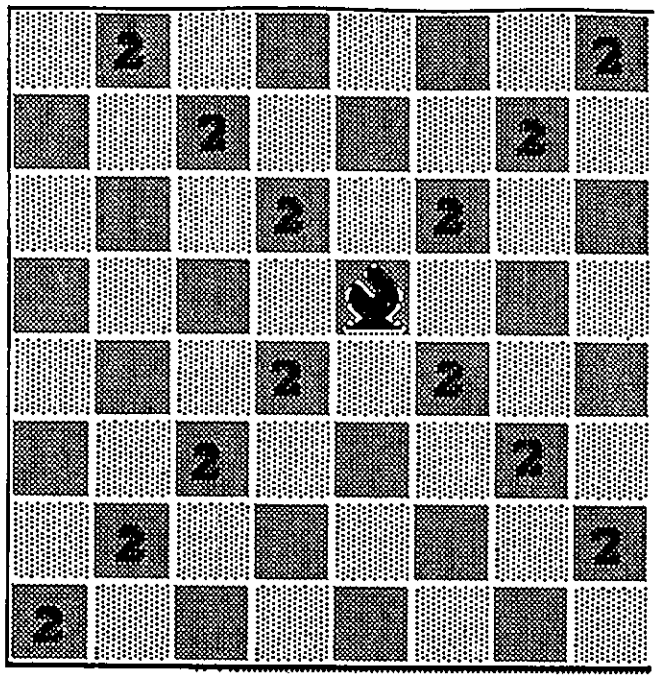


Figure A.3: The direct radiant power of black bishop

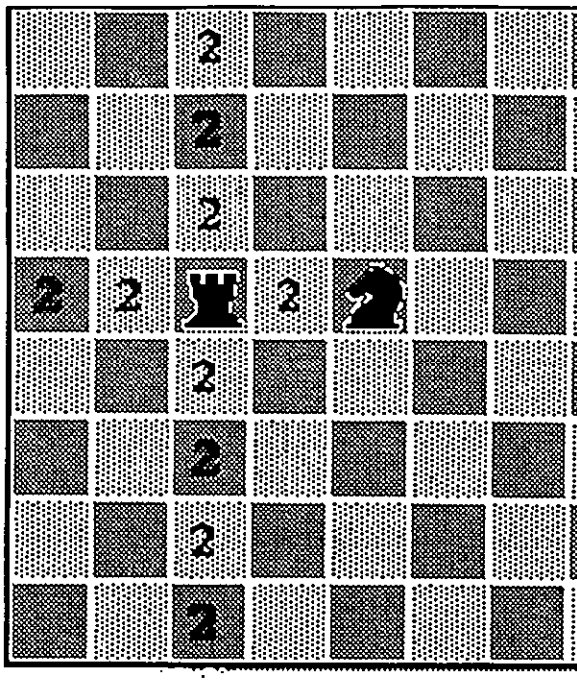


Figure A.4: The direct radiant power of black rook

From these images, we understand that chess can follow the same principles as those used in Go. The difference is that the P in chess does not reduce with the distance. Why not? Because chess is not a miniature of the real world and its rules are not natural. The chess rules permit the pieces (for example: a rook) to move one square or several squares at the same time.

Do we need to calculate IP (Indirect Power) and can we calculate it ? Both answers are yes. Figure A.5 to Figure A.8 show you the DP and IP radiated by the corresponding piece.

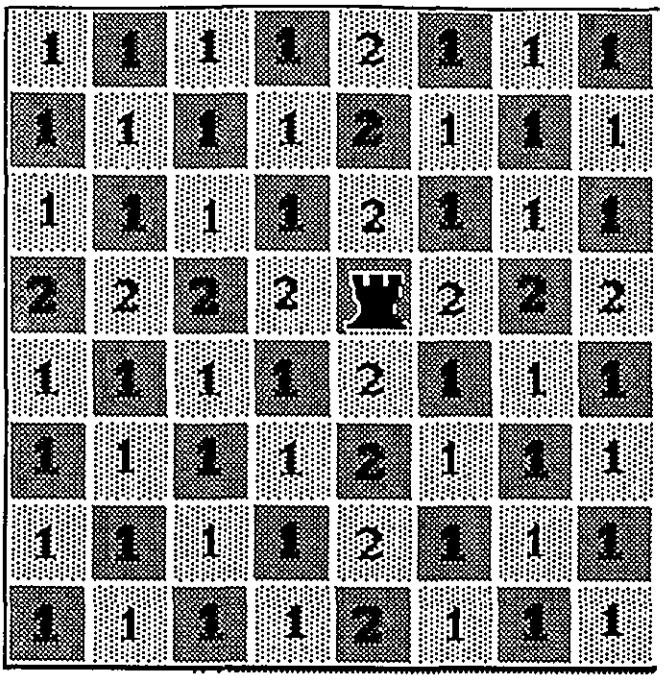


Figure A.5: The power of black rook knight

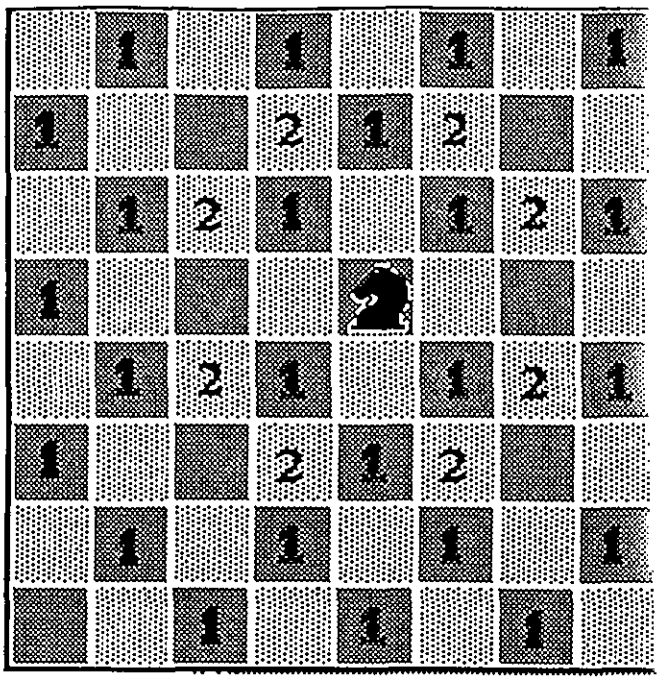


Figure A.6: The power of black

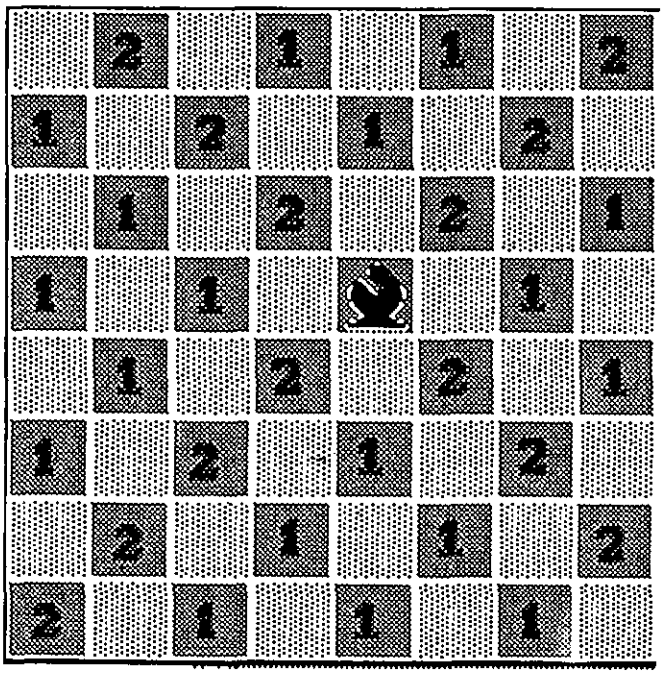


Figure A.7: The power of black bishop

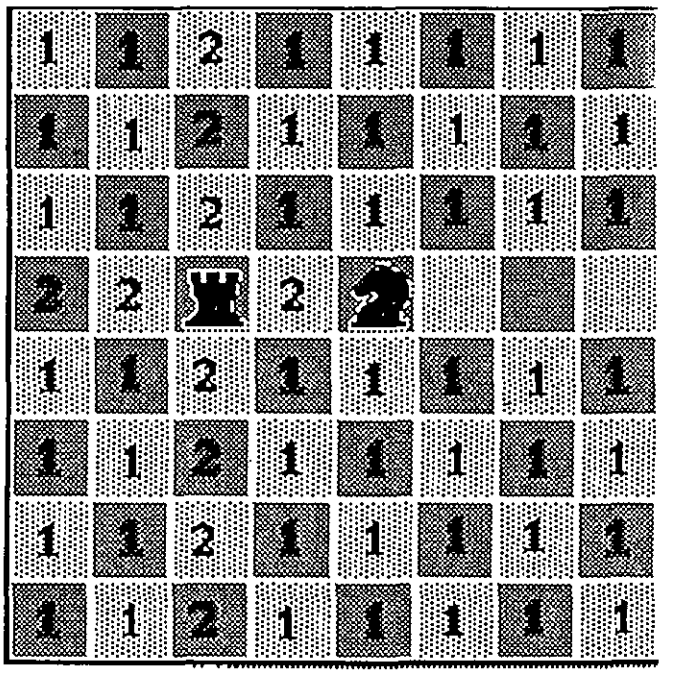


Figure A.8: The power of black rook

How do we calculate the power of a chessman in a chess-board square ? To do this, we have to first define a concept "distance":

distance(chessman, square) = minimum moves from the chessman to the square

and we take the formula (A.1) to calculate a chessman power in a square:

$$\text{power(chessman, square)} = \lfloor P * AR^{\text{distance(chessman, square)} - 1} \rfloor \qquad \text{(A.1)}$$

then

$$\text{SumPower(square)} = \sum_{i=\text{all chessmen}} \text{power(i,square)} \qquad \text{(A.2)}$$

In the examples of Figure A.1 to Figure A.8, we have chosen P = 2 and AR = 1/2. Of course we could choose other values.

References

[AGJ] The American Go Journal. American Go Association. P.O.BOX 397, Old Chelsea Station, New York, NY 10113.

[Boon 89] A Pattern Matcher for Goliath. M. Boon [CG] 13 (Winter 1989-1990), 12-23.

[Chen 89] Group Identification in Computer Go. K. Chen *Heuristic Programming in Aritifical Intelligence, The First Computer Olympiad* (eds. D.N.L. Levy & D.F. Beal), pp. 195-210. Ellis Horwood, Chichester.

[CG] *Computer Go.* D.W. Erbach (eds), 71 Brixford Crescent, Winnipeg, Manitoba R2N 1E1, Canada

[Kierulf 90] Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies. A. Kierulf *Ph.D. Thesis, ETH Zürich* (1990).

[Knuth & Moore 75] An analysis of alpha-beta pruning. D.E. Knuth; R.W. Moore *Artifi. Intell.* 6, (1975), 293-326.

[Levy 88] *Computer Games II.* D.N.L. Levy (eds), Spring-Verlag, New York Inc, 1988.

[Levy 89] *Heuristic Programming in Artificial Intelligence - The First Computer Olympiad.* D.N.L. Levy; D.F. BEAL (Eds.) Ellis Horwood Ltd., Chichester, West Sussex, 1989.

[Ryder 71] Heuristic Analysis of Large Trees as Generated in the Game of Go. J.L. Ryder *Ph.D. Thesis, Standford Univ.*, (1971), 1-298.

[Shannon 50] Programming a computer for playing chess. C.E. Shannon *Philosophical Magazine* **41**, 314 (1950), pp. 256-275.

[Wilcox 78-84] Computer Go. B. Wilcox [Levy 88], 94-135.

[Wilcox 87] Issues in Joseki processing. B. Wilcox [CG] 4 (Fall 1987), 18-23.

[Zobrist 69] A Model of Visual Organization for the Game fo Go. A.L. Zobrist *Proc. AFIPS 1969 Spring Joint Computer Conf. 34* (Boston, Mass., May 14-16, 1969), 103-112. AFIPS Press, Montvale, NJ, 1969.

[Zobrist 70] Feature Extraction and Representation for Pattern Recognition and the Game of Go. A.L. Zobrist *Ph.D. Thesis, Univ. of Wisconsin*, 1970, 1-152.