**9x9 Computer Go**

Msc Computer Science Summer Project 2011/2012

Supervisor: Mark Herbster

**Name: Xiying Huang**

Date: 04/Sep/2012

*This report is submitted as part requirement for the MSc Computer*

*Science degree at UCL. It is substantially the result of my own work*

*except where explicitly indicated in the text.*

*The report may be freely copied and distributed provided the source*

*is explicitly Acknowledged.*

**Abstract**

Go is one of the most famous board game in the world. With a history of more than 4000 years, Go is a decent board game originated in ancient China. Although Go game has a long history, computer Go has not yet been developed more than 100 years. The AI of computer Go game has fast developed in recent years but not enough. The best Go programmes are ranked only as 2 – 5 dan today. The AI of computer Go beat the best Go player has long way to go.

The aim of this project is made a 9x9 computer Go game which has an AI that can satisfy most of the beginners of Go game. The difficulties of this project are to achieve the rules of Go game on board and build up an artificial intelligence (AI).

The AI in this project is used the mini-max search way that based on the Monte Carlo method. Monte Carlo method is the most basic and famous algorithm for the Computer Go game. This method is also useful in the 9x9 size board. It can not solve all the problems of the computer Go game but provide enough data to analyse and help people to build up a new AI algorithm.

# Contents

1. **Introduction**
   1. Overview………………………………………………………………….5
   2. Aims………………………………………………………………………6
   3. Problem Specification……………………………………………………..6
   4. Report Structure……………………………………………………………7
2. **Background**
   1. History of the Go Game……………………………………………………8
   2. Rules of the Go Game………………………………………………………9
      1. Liberty…………………………………………………………………9
      2. Life and Death………………………………………………………..10
      3. Ko……………………………………………………………………..11
      4. Victory………………………………………………………………..12
   3. Complexity………………………………………………………………..12
   4. Software…………………………………………………………………..13
3. **Implementation of Interface on Java**
   1. Interface…………………………………………………………………...14
4. **Algorithmic of Go Rule**
   1. Counting Liberties…………………………………………………...........16
   2. Capturing Stones………………………………………………………….17
   3. The Unavailable Point…………………………………………………….19
      1. Suicidal step………………………………………………………….19
      2. Ko-fight……………………………………………………………….20
   4. End of Game and Scoring…………………………………………………20
5. **Machine Learning Applied to Go**
   1. Evaluation Function……………………………………………………….21
      1. Best-first Search………………………………………………………21
      2. Adversary Search……………………………………………………..21
   2. Mini-max of Tree Search………………………………………………..22
   3. Using a Monte Carlo Evaluation
      1. Hold the Eyes………………………………………………………24
      2. Finish the Game……………………………………………….……25
      3. Decided the Winner………………………………………….……..26
      4. Calculate the Value…………………………………………………27
   4. Result of AI………………………………………………………..…….28
6. **Evaluation and Conclusion**
   1. Evaluation………………………………………………………..………28
   2. Conclusion……………………………………………………..…….…..28

**Appendix A System Manual**

**Appendix B Code List**

**References**

**Chapter 1**

**Introduction**

* 1. **Overview**

Go is one of the most famous board games in the world. A large number of people play it every day. But the board and the stones are too inconvenient to carry, so the computer Go started to appear. Since the 1960s, a lot of computer Go games have been developed. In addition to achieving the basic rules of the Go game, computer can do many more things with the advanced technology like to judge the game quickly, to save and replay a game step by step.

Like many other board games, the artificial intelligence (AI) of the computer Go game has been developed in recent decades. However, the level of the best AI today can not be compared to the professional Go players. It is not only due to the speed of the computer or the ability of the algorithm. The target of the Go game and the size of the board(19x19) create the difficulty to find out the best next step.

Monte Carlo method is the most basic and famous algorithm for the Computer Go game. It can not solve all the problems of the computer Go game but provide enough data to analyse and help people to build up a new AI algorithm. Many of the top computer Go programmes are based on Monte Carlo method.

* 1. **Aims**

The aim of this project is making a 9x9 computer Go game. There are two modes of this computer game: “two players” and “play with computer”.

For the interface, it must achieve all the rules of Go game. This programme is not only a board that can be put on stones. The system can take off the captures and check unavailable points automatically, no matter the player is a beginner or a person with no knowledge of Go.

In “play with computer” mode, the player can play with the AI in this programme. With the mini-max search way and the Monte Carlo method, the programme can calculates the best next step. The level of the AI can satisfy the purpose of playing with most of the beginners of Go game.

* 1. **Problem Specification**

The difficulties of this project are to achieve the rules on board and build up an artificial intelligence (AI).

For human beings, the rules of Go can be learned quickly. However, to implant the rules in the programme it is not an easy task. From the experience of other predecessors, use the Breadth-first tree search to count the liberties of a portion of the stones or find the captures is a good choice in this situation.

Go is not the same as chess. The chess just have one objective, with is checkmate. The algorithm is built all around this objective. But the objective of Go is try to capture more territory than your opponent. For the Go players, thinking of how to capture the other one’s stones, how to keep their own stones alive, how to get more areas are always their concern in every round. However an AI does not operate like a human brain. If it wants to have a perfect AI, the algorithm will be really complex. The best way to do that is using marching learning. Calculating every point’s percentage of winning and choosing the best one are the most used functions in the domain. Monte Carlo method is a very useful solution to the computer Go game. The board in this programme is 9x9, not very big. With the Monte Carlo method and the Mini-max algorithm, the AI in this project can be strong enough.

* 1. **Report Structure**

The remainder of this report is structured as follows:

Chapter 2 is the background of the computer Go. Describes the history of computer Go, the rules of Go, the complexity of AI, and the platform of this programme been build.

Chapter 3 is the work that to implement the interface of the computer Go game

Chapter 4 describes the algorithmic to implement the rules of the Go game.

Chapter 5 describes the theory and implementation of AI in computer Go.

Chapter 6 evaluates the programme and gives the future work.

**Chapter 2**

**Background**

* 1. **History of the Go Game**

With a history of more than 4000 years, Go is a decent board game originated in ancient China. In China it is named “Weiqi”, “Go” is the name from Japanese. Japanese introduced the Go game into western world from 16th century. This is the reason that why that most of the terms of Go is transliterations from Japanese.

Although Go game has a long history, computer Go has not yet been developed more than 100 years. The earliest work in the domain of Computer Go was developed in 1962 and the first complete programme was written by Albert Zobrist in 1968[1]. This first computer Go game used the pattern recognition and can detect “ko”. In 1980s, go game entered a period of rapid development. With the popularity of affordable PCs, the new programmes springed up like mushrooms after rain. Some famous programmes we know today such as *Dragon[2]*and *Golath[3]*was made during that times. *Mogo[4]*, one of the famous programme recent years was made in 2006. It was based on the recent developments in Monte Carlo Tree Search and giant leap in the evolution of the AI of the computer Go game. Today, the best Go programmes are ranked as 2 – 5 dan(level of expert player)[5] and the top programmes play competitively against professionals with around a 7 stone handicap [6].

The computer Go game is much easier to reach a higher level on 9x9 board, than on a bigger board and it has more variants than a smaller board. There are lots of 9x9 computer Go championships all over the world. Most of the programme makers are students in the university. The programmes they made are not for commercial use, it is a way to communicate with others, to make new friends, to learn some new ideas and a challenge to human’s brain.

* 1. **Rules of the Go Game**

In different countries, the rules of Go game are not exactly the same. In this project, the programme uses the Chinese rules of the Go game.

The board for Go game has a different size. The normal size board for advanced players is 19x19. The 9x9 size board is for the beginner. The stones for Go game are of two different colours: black and white. Either the black stones or the white stones are all the same. Every stone need to set on a cross of the board. The black always sets first and then each colour takes turns.

* + 1. **Liberty**

On a board, an unoccupied cross beside a stone is called a “liberty”.

In China, the liberty is called “Qi” (means air). No one could live without air, so do the stones in Go. If a part of stones does not have any liberty left, they will “die”.

The liberty of a single stone on the board is shown below:



Figure 2.1 The liberties of a single stone

As is shown, the stone in the corner has only two liberties. The stone on the edge has three liberties and the one in middle has four.

If the stones of the same colour were linked together, the liberty should be count together. In figure 2.2, these two stones have 6 liberties.



Figure 2.2 The liberty of two stones

The rules, encourage more stones to be linked together for more liberties. In this case, the other colour’s stones will have difficulty to capture them.

* + 1. **Life and Death**

If a part of stores does not have any liberty they will “die” and be taken off from the board. This is called a “capture”. The captures need to be taken off from the board and the empty points which after the capture can be used in the game later on.

An unoccupied point which can not be set a stone on is called an “unavailable point” or “unavailable step”. If the player set a stone on, this stone will not have any liberty. These points are not available for the next step.

For example, in Figure 2.3 if a white stone is set on point ‘A’, these three white stones can not have any liberty. So the white stone player must not set the next step in point ‘A’.



Figure 2.3

There is another situation, if a point is available for a capture in the next step it is not an unavailable point.

The same in Figure 2.3, if black stone player set a stone on point ‘A’, there is also no any unoccupied places beside it. However, the two white stones beside it will be captured. Then the black stone in point ‘A’ will have two liberties after this step.

An enclosed liberty (or liberties) is called an “eye”. If a portion of stones has two or more eyes, they will not be captured. Because, no matter how many steps the other colour set, there must be two unavailable points inside this portion of stones.



Figure 2.4

In Figure 2.4, the points “a” and “b” are two eyes for the black stone player. Both ‘a’ and ‘b’ are unavailable points for the white stone player. So that, the white stone player will never be able to capture this portion of the black stones.

* + 1. **Ko**

“Ko” is a situation in the Go game.

Figure 2.4 Figure 2.5

Figure 2.4 and 2.5 is showing a Ko. If the white stone player set stone on point ‘A’ the situation will become Figure 2.5. After that if the black stone player set a stone on point ‘B’, the shape will become Figure 2.4 again. If both of the players repeat these moves again and again, the game can not be continued.

The rule of Ko is to solve this problem. For example in Figure 2.4, the white stone player put on point ‘A’ to capture one black stone and the situation become to Figure 2.5. After that the black stone player is not allowed to put a stone on point ‘B’ to re-capture the white stone, it must set step on elsewhere. Then the white stone player can set step on ‘B’ to end the Ko, or respond to the previous black step. If the Ko is not ended, the black stone player can put a stone on ‘B’ to capture the white stone back and let the situation become to Figure 2.4 again. And then it is the turn of white stone player to try to find a point elsewhere on the board that can make the black stone player respond. If black respond to that, white can retake the Ko. A repetition of this is called “a Ko fight”.

* + 1. **Victory**

If both of the players pass the next step, the game will be finished.

The victory rule for Go game is different in different countries. When a Go game is finished, the black has to compensate the white for the advantage of playing first. This is done by adjusting the final score by a value, usually between 5.5 and 8.5 points (in 19x19 board). Because of the complexity of the Go game, the exact value is hard to count for the advantage of the first player. And the half point is to prevent draws. [7]

In Chinese rule, a player’s score is the number of stones it has on the board and plus the number of empty intersections surrounded by that colour’s stones. And the adjustment for Chinese rule is 7.5 points. So, if the score of the black is 7.5 points higher than white, the winner is the black side, and vice versa.

For 9x9 board, the adjustment does not have a unified rule in the world. In this project, the adjustment is debug by thousands rounds of Go games by the programme to minimise the possibility of the error. The more information is provided in 5.3.3.

* 1. **Complexity**

Go is not like chess or Five-in-Row. In these board games, they only need to achieve the move or put the stone on the board in the programme. However, for Computer Go game, the situation after the next step must be considered. The system need to confirm that the next step is not on an unavailable point, is not re-capture in Ko fight. And after a step set, the system needs to count the liberties for every portion of the stones which may lose their last liberty and take all of the captures off from the board. These things sound easy to do, but using a programme to achieve them is complex.

For the AI in computer Go, the board of Go game is too big. In the 19x19 board, the first step has 361 kinds of choices, and in the 9x9 board it has also 81 kinds of first step. That means, if a 9x9 computer Go game programme is used mini-max algorithm to calculate the next three steps and choose the best next step, the system needs count 813 times. In the chess game, it has only 20 kinds of first step. And with the process of a game, the best next step for chess game will become easily to be found. However in a Go game, the algorithm will become more and more complicated with the process of the game.

* 1. **Software**

The whole programme is made with “Netbeans IDE 6.9.1”. Netbeans can run the code step by step. It is very useful when the maker debug the programme. The large database in the Netbeans can make the maker conveniently to find a useful function or write a new function. It is really useful software to compiler Java language.

The icons in the programme and all the figures with Go game in this report are drawn with “Adobe Photoshop CS3”. The materials of the figures of Go game are taken from the screenshot of the programme in this project.

**Chapter 3**

**Implementation of Rules of Compiling**

* 1. **Interface**

In this project, the interface is built with the “JFrame” and “JButton” in java. Every cross on this Go board is a small button. The programme also has a 9x9 digital board to keep the state of Go board. The system uses the digital board to calculate the situation after a step then show them on the image board for the player.

**Board**

The most popular way to build a board by computer is setting every point as a small picture on the board. For example in a chess game, with the process of the game, the empty point will be set chess on and the chess may be moved to the other place to make this point empty again. For the player, it is seems that the chess is moved but for the programme, it is only the images being changed in these points.

It is the same in the Go game, every cross is a button. When the player clicks a button, the system will respond to the click in the way of changing the picture on the board to a picture of stone.

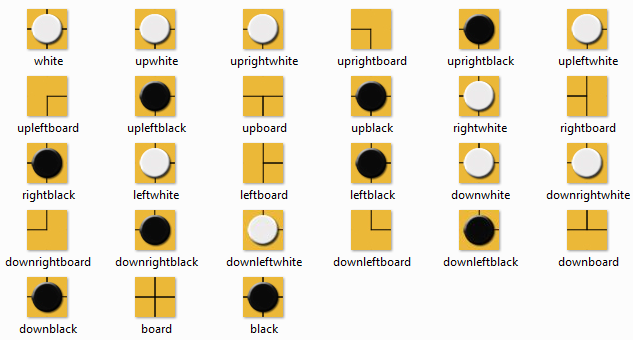


Figure 3.1 The Pictures of board been used in this programme

On the other hand, the board of Go game has four corners and four edges. The rule of the stones in the corners or on the edges is different from the rules with the other stones. So for the board in a computer Go programme, not only the different pictures in the corners and edges are needed, but also the data in the corner and edges. These data need to be processed in a different way. The board has four different corners and four different edges, that means for every click from the player, the system needs to confirm nine times to find the correct picture to set.

In this project, there are two arrays to save the state of the board. “board[][]” is the array which saves and shows the picture for the player. And “boardInt[][]” is the array that works for the system. The system saves the state of every point in “boardInt[][]”, ‘1’ means black, ‘-1’means white, ‘0’means empty, ‘10’means out of the board. After a click, the system will used the data in array “boardInt[][]” to calculate, and then send the result to the array “board[][]” to show the result to the player. This presents the communication between the player and the system in this programme.

The full program is in the Appendix B.

**Chapter 4**

**Algorithmic of Go Rule**

This chapter only shows the rules which can be achieved in this programme. Go game is very complex. There are large numbers of situations that can not be solved with the basic rules. For example, if two portions of stones become a “Seki” [8], both of them can not adopt the simple “life and death” rules to deal with their situation. A “Seki” situation is shown in Figure 4.1. If someone set a stone on “A”, the other colour will set a stone on “B” to capture a portion of stones. For this reason, both of the players will set a stone neither on “A” nor on “B”. This situation is called “Seki”.

In this project, the board size of Go game is limited to 9x9. That is to say, the “Seki” or other special situations on the board are small probability events. As the result, the rules which is not related to this project will not be discussed in chapter 4.



Figure 4.1 a “Seki” Situation

* 1. **Counting Liberties**

In this system, the way to count the liberty of the stones is to traverse every point on the board and save the number in an array. When the system need to decide whether a portion of stones has been captured or not, it will call this function then sum up all the liberties of the stones which are in this portion. If the number of liberty which has been added is zero, that means this portion of stones is captured. It is also used to check the unavailable points. However, this way can not find out the correct liberty of a group of stones. In most of the situations, one liberty will be counted for several times in this system. In this programme, this function is only used to confirm the captures and to prevent setting a stone on unavailable points. Because, if the liberty is zero, no matter how many times the system calculates, the number is always zero.

The name of the function in this programme is “stoneLiberty()” and the array which is used for saving the number of liberties is named “liberty[][]”.

point(x,y)

if (point in the corner)

liberty[x][y]=2;

if (a cross near point is not empty)

liberty[x][y]--;

if (point in the edge)

liberty[x][y]=3;

if (a cross near point is not empty)

liberty[x][y]--;

else

liberty[x][y]=4;

if (a cross near point is not empty)

liberty[x][y]--;

The full program is in the Appendix B p34.

* 1. **Capturing Stones**

After one step, if a stone has no liberty, it will be captured and needs to be taken off from the board.

In this programme, the system will count the liberty of the portions of stones behind last step. If the liberty of one of the portions of stones is zero, this portion of the stones will be taken off by the system.

The way to achieve this function is using Breadth-First-Search. In the simplest way, with one first point, system will search the surrounding four points to check the state of them, and then search the next four points around these points which in the same state with the first point. The point being checked already will not be searched again. This action is repeat until the system can not find a point which has the same statement as the first point. After the search is finished, there is a list of stones that are in the same portion (or a list of an area with unoccupied points). The part of the code is shown below:

Int begin = 0, end = 0;

point (x,y);

int newx=x, newy=y

do

{

findCount=0;

for (int i=begin; i<=end;i++)

check the points surrounding point(newx,newy);

begin = end+1;

end = end + findcount;

}while (findCount>0)

The full program is in the Appendix B p36.

In this case, the system will count the liberties of the portions of stones which have the chance of being captured. If it is zero, this portion of stones must be captured and be taken off from the board. After one step, there are at most four portions of stones that can be captured. For example in Figure 3.1, if the white stone player set a stone on point ‘A’, the four portions of black stones named ‘a’ ‘b’ ‘c’ and ‘d’ around point ‘A’ are all captured. For this reason, the system will use the Breadth-First-Search four times after one step to achieve “removal of the captures”.



Figure 3.1

This programme uses three functions to solve this case, “arrayList checkEat()”, “arrayList findEatPart()” and “eat()”. “findEatPart()” is a function that is used to achieve the Breadth-First-Search algorithm. The function “checkEat()” is used to confirm that “findEatPart()” is run four times for different points and save all the captures in a list. If the list of captures is empty, the programme will stop and wait for the next order. If not, the function “eat()” will be called to take the captures off from the board. Not only the images of the captures in board will be changed, but also their states on the digital board will be changed to zero.

* 1. **The Unavailable Point**

In Go games, there are two kinds of unavailable point in the Chinese rule.

One is the “suicidal point”. If a player set stones on that point, the portion of stones after this step will not have any liberty. The other one is in “Ko” fight. (Detailed information for these rules is explained in the Chapter 2 of this report.)

* + 1. **Suicidal step**

With the purpose of checking whether a step is a suicide step or not, this programme as well uses the Breadth-First-Search algorithm. Assume that the step can be set and find out the portion of stones which it is in. Then count the liberty of this portion of stones. If it is not zero, that means this step is not a suicidal step, the system will set a stone on for this step. If the liberty is zero, the system will check if this step can capture any stones. If possible, this step is also not a suicidal step.

If a step that will leave a portion of stones with no liberty and can not get any captures, this step is a suicidal step and must not be set on the board.

In this programme, there are two functions to solve this problem, “boolean checkAvailable()” and “arrayList checkEat()”. The “checkAvailable()” is used to check the liberty after the assumed step. If the number of liberty is not zero, the function will return “true”, and then the system will set the stone on that step. If it is zero, the function will return “false” then call the function “checkEat()”. If the array list which is returned by “checkEat()” is empty, this step must be unavailable. The system will not set any thing on board and wait for the next order.

The full program is in the Appendix B p40.

* + 1. **Ko-Fight**

With the purpose of checking a situation to be a Ko or not, it needs to fulfil three conditions. First is a capture which can be taken after the step. Secondly, after this capture, the set stone in this step only has one liberty. The last one is that, there is no stone which has the same colour surrounding the set stone in this step.

It is easy to achieve the Ko fight in the programme. Set a Boolean global variable named “isKo” to check the last step was on a Ko or not. If a step captures one stone, after that has only one liberty and has not any same colour stones surrounding it, the value of the “isKo” will be changed from “false” to “true”. If the value of the “isKo” has already been “true”, that means the last step was set on Ko, then the system will not allow this step to be set on the board. The player has to set a stone on the other point to change the value of the “isKo” back to “false”. Then the Ko fight has been achieved.

if (stonesDead.size()==1 && liberty[x][y]==0)

isKo = true;

else

isKo = false;

* 1. **End of Game and Scoring**

In this programme, there is a “Finish” button in the menu of the interface. When the player confirms to finish a game, the system will count the score for both colours and decided which one is the winner. However it is not a perfect function to decide the winner. If it is an unfinished Go game, the result from this function will different from reality. It is a very complex problem to count the score precisely. Every programme will make mistake in this part, even the best software of computer Go game in the world. On the other hand, in most of the time, the function which is used to decide the winner in this programme is in service of the random games with Monte Carlo method. The more information about this function is described in 5.3.

**Chapter 5**

**Machine Learning Applied to Go**

First of all, to design an AI, the programer has to decide a method. Using a tree search to find out the best result is the basic way. In Go game, for a 9x9 size board, there are 81 points on a board. With the blind search, there are 81! nodes in the search tree and either depth-first or breadth-first search have to be search all of the nodes. The blind search is not possible to be used in this situation.

* 1. **Evaluation Function**

The evaluation function is to solve the time-consuming problem in blind search. The way of searches which use the evaluation functions is named heuristic search. With the heuristic search, not all of the nodes are required to be searched. Different functions have different tendencies. The better result can be found by choosing a correct function.

* + 1. **Best-first Search**

The best-first search is an algorithm which combines the advantages of both depth-first and breadth-first search. With this search, the system can adopt the best node in this step and generate this branch. Then repeat this way step by step. This search algorithm can save a larger number of searching time compared with the blind search.

However, the best-first search only considers the next step. For an AI used in the two-player game, the steps of the opponent also have to be considered.

* + 1. **Adversary Search**

The adversary search is used in a situation with two or more agents. In this algorithm, there is not necessary to find a goal. For this reason, the time used in this search way will be not too long. On the other hand, the system using this way of search has to findout all possibilities in advance then workout the best bunch. Because of this reason, the game that is played by two competitive players is adoptable to this algorithm, especially the board game. For example, one step for chess has a limited number of choices to be analysed, and then the best step will be found out.

The perfect adversary search for deterministic games is the mini-max search. The machine learning system in this project employs the mini-max algorithm.

* 1. **Mini-max Tree Search**

Mini-max is a heuristic search for minimizing the possible loss for a worst case scenario. Comparatively, this search tree can find the possible winning way.

The way to apply this search is used a function to calculate the value of the nodes which after “n” steps. Then return a value to the parent node which is one level higher above. It is returned a minimum in MIN layer and a maximum in MAX layer for every bunch. After all, the only one value can be returned to the final parent node. The bunch which returned this value is the best next step.

MAX

MAX

MIN

30

50

80

10

40

50

30

60

20

30

10

20

30

Figure 5.1 a simple three steps mini-max search tree

In this project, there is a three-step searching thee that can be built in computer turn. Two arrays are used to implement this algorithm. They are named “mark[][]” and “position[][]” in the program. There is a simple table Figure 5.2 to describe the processing in this program.

At first, use the Bubble Sort to find the maximum number of the value of node a1, a2 … This number is the value for node a. Then save this number in the array “mark[][]” and the position of the node which have this value is to save in array “position[][]”. All of the nodes in layer 3 are processed with this step. After that, the “mark[][]” is full of the value of the nodes in layer 2. Use the Bubble Sort again to find out the minimum in array “mark[][]” the result is the value for node A. The next step is to look for the position of the node which has this value with the array “position[][]”. Finally, the best next step is found.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Layer 1 (max) | Node A | | | | | | |
| Layer 2 (min) | Node a | | | Node b | | Node c… | |
| Layer 3 (max) | Node a1 | Node a2 | … | Node b1 | … | Node c1 | … |

Figure 5.2

* 1. **Monte Carlo method**

For the searching tree with the mini-max algorithm, it needs a function to calculate the values.

The Monte Carlo method is a Randomized Algorithm that relies on random evaluations to solve problems approximately.[9] It is used in computer or mathematical systems which can not be exactly calculate.

Since the 1980s, people began to do research on the machine used in the computer Go game. The most popular way is using Monte Carlo method to set a large numbers of random games and calculate the winning percentage for every possible next step. The point which has the highest winning percentage may be the best next step.

However, this method can not be applied to the normal size board (19x19) directly, since it is time-consuming and non-efficient. For example, a system sets 100 rounds of random games for one point and in every round there are 300 random steps, in other words this system needs to set 106 steps to decide one step and the result can almost be random.

Nevertheless, the Monte Carlo method can be directly used in 9x9 board. Since this method can be adopted in most of the situations. What is more, this method only calculates the winning percentage by computer. In other words, the calculating result which has been found by the system will not be affected with the Go level of programme makers.

For these reasons, the function is used for mini-max search in this program is built with Monte Carlo method.

* + 1. **Hold the Eyes**

The biggest problem to achieve Monte Carlo method is to make sure that the system will set the stone in an “eye” by a random choice properly. It is simply because to set a stone inside an eye is not an unavailable step. As is said before (in 2.2.2), if a portion of stones has two or more eyes, they will not be captured. However, if a portion of stones has two eyes and the player of this colour fill in one eye, then this portion of stones will be captured in the game later on. In a random game, if the system is lack of the limitation to the steps which fill in eyes, the game will never be finished.

This programme saved three kinds of situations of eyes which be shown in Figure 5.3. All of the points which are mark ‘a’ in the figure 5.3 are eyes. If the system confirms that the next random step is fill in an eye, this step will not be allow to set and the system will turn back to random a new point.

Figure 5.3

With this system, the programme will hold most of the eyes on the board.

* + 1. **Finish the Game**

In the real practise of Go game, if both of the players pass the next step, the game will be finished. The same way is used in the computer Go game. In a random game, if the system cannot find an available next step, it will set a flag of pass for the next step. When both of the colours set a flag for the same step, this round of random game is finished.

The workflow for this programme is shown below:

Get a random point

Start

Filling an eye?

Unavailable point and m<20 m++

Set m=0, n=0

Available point

Yes

n<10 n++

Set this step on board

Set “pass” = false

No

Yes

n>=10

End

Check the value of “pass”

Unavailable point and m>=20

Change the colour

false

set “pass” = true

true

Figure 5.4

With this function, in most of the times it can only leave the “living” stones on the board.

* + 1. **Decided the Winner**

In the Chinese rule, the captures are not used to count the score. As it appears in 2.2.4, the score is made up with the number of stones it has on the board and then plus the number of empty intersections surrounded by that colour’s stones.

In a random game, when a round is finished, all stones on the board are “alive”. Then, there are five states of point on the board: black stone, white stone, black area point, white area point and neutral point (the point neither in black area nor in white area). Sum of the number of black stones and the number of black area points is the score for the Black, the same way to calculate the score for the White. By comparing these two numbers, the system can know which colour is the winner.

As is said before, the adjustment in 9x9 Go game does not have a unified number in the world. With the purpose of minimising the possibility of the error, testing enough numbers of random games which begin with empty board is necessary in this project.

The winning percentage of 1000 random games:

|  |  |  |
| --- | --- | --- |
| Adjustment | Black Win | White Win |
| +0 | 53.5% | 45.6% |
| +0.5 | 51.1% | 48.9% |
| +1 | 49.4% | 49.7% |
| +1.5 | 45.9% | 54.1% |

(If the adjustment is an integer number, there is possibility of draw game, so the sum of the percentage in the form is less than 100 %.)

Figure 5.5

As is proved in the table, the nearest adjustment is +1. Therefore, an extra 1 point will be added to the score of the white.

The full program is in the Appendix B p42.

* + 1. **Calculate the Value**

After a step, the system will run a number of random games and then count the rounds of winning, then there is a number that can demonstrate the mark of this step. The higher the mark reaches, the higher of winning percentage it will be. Therefore, this mark can be the value of this step to be used in the mini-max searching tree.

* 1. **Result of AI**

After achieving the success the mini-max search that based on Monte Carlo method, a basic AI has been completed.

However the program in this project is not well enough. The completed program and the final result will be presented in the demonstration.

**Chapter 6**

**Evaluation and Conclusion**

* 1. **Evaluation**

The interface of the program in this project is very successful that most of the targets have been completed.

The theory of AI is almost enough. Then after debugging the program, there will be a 9x9 computer Go game software that can satisfy most of the beginners of Go game.

The only drawback of this project is the running time of the AI is too long. On a conservative estimate, the AI in this project will use at least one minute to calculate one step.

* 1. **Conclusion**

There are two major parts of this project. One is to implement an interface for a computer Go game. The other is to build an AI of Go.

The interface is not too hard to design, but only have idea can not implement all of the rules. Debugging the program step by step carefully is the key point to success.

For the AI, use mini-max search way with the Monte Carlo method is a challenge. The algorithm needs more test and adjustment to save running time in the future. However it is affine program.

**Appendix A**

**System Manual**

The project’s source code can be build in ‘NetBeans’.

Minimum Hardware Configurations:

Microsoft Windows XP Professional SP3/Vista SP1/Windows 7 Professional:

Processor: 800MHz Intel Pentium III or equivalent

Memory: 512 MB

Disk space: 750 MB of free disk space

$ The code is in …\GoGame\src\gogame\main.java

**Appendix B**

**The public variable**

private static final long serialVersionUID = 1L;

static int boardLineCount = 17;

private JButton board[][] = new JButton[boardLineCount][boardLineCount];

int boardInt[][] = new int[boardLineCount][boardLineCount];//the statement of the stones, black is '1', white is '-1', empty is '0'

int testChess[][] = new int[boardLineCount][boardLineCount];

int userFlag = 1;//current color, black is '1', white is '-1'

int liberty[][] = new int[boardLineCount][boardLineCount];

int pointInt[][] = new int[boardLineCount][boardLineCount];

int pointTestInt[][] = new int[boardLineCount][boardLineCount];

int stones[][] = new int[200][2];//to save the coordinate of every step

int steps=0;//the number of step

int comWin=0;

String winner;

boolean pass = false;//ture means one color pass

boolean userType = true;// ture Man false PC

boolean comTurn =false;//ture is PC's turn, false is player

boolean isKo = false;//ture is Ko, false is not

/\*set the Icons \*/

String direction = "./src/gogame/Icons/";

ImageIcon beginIcon = new ImageIcon(direction + "board.jpg");

ImageIcon whiteIcon = new ImageIcon(direction + "white.jpg");

ImageIcon blackIcon = new ImageIcon(direction + "black.jpg");

ImageIcon rightIcon = new ImageIcon(direction + "rightboard.jpg");

ImageIcon rightWhiteIcon = new ImageIcon(direction + "rightwhite.jpg");

ImageIcon rightBlackIcon = new ImageIcon(direction + "rightblack.jpg");

ImageIcon leftIcon = new ImageIcon(direction + "leftboard.jpg");

ImageIcon leftBlackIcon = new ImageIcon(direction + "leftblack.jpg");

ImageIcon leftWhiteIcon = new ImageIcon(direction + "leftwhite.jpg");

ImageIcon upIcon = new ImageIcon(direction + "upboard.jpg");

ImageIcon upBlackIcon = new ImageIcon(direction + "upblack.jpg");

ImageIcon upWhiteIcon = new ImageIcon(direction + "upwhite.jpg");

ImageIcon downIcon = new ImageIcon(direction + "downboard.jpg");

ImageIcon downBlackIcon = new ImageIcon(direction + "downblack.jpg");

ImageIcon downWhiteIcon = new ImageIcon(direction + "downwhite.jpg");

ImageIcon upRightIcon = new ImageIcon(direction + "uprightboard.jpg");

ImageIcon upRightBlackIcon = new ImageIcon(direction + "uprightblack.jpg");

ImageIcon upRightWhiteIcon = new ImageIcon(direction + "uprightwhite.jpg");

ImageIcon upLeftIcon = new ImageIcon(direction + "upleftboard.jpg");

ImageIcon upLeftBlackIcon = new ImageIcon(direction + "upleftblack.jpg");

ImageIcon upLeftWhiteIcon = new ImageIcon(direction + "upleftwhite.jpg");

ImageIcon downRightIcon = new ImageIcon(direction + "downrightboard.jpg");

ImageIcon downRightBlackIcon = new ImageIcon(direction

+ "downrightblack.jpg");

ImageIcon downRightWhiteIcon = new ImageIcon(direction

+ "downrightwhite.jpg");

ImageIcon downLeftIcon = new ImageIcon(direction + "downleftboard.jpg");

ImageIcon downLeftBlackIcon = new ImageIcon(direction + "downleftblack.jpg");

ImageIcon downLeftWhiteIcon = new ImageIcon(direction + "downleftwhite.jpg");

ImageIcon boardBPointIcon = new ImageIcon(direction + "board\_bPoint.jpg");

ImageIcon boardWPointIcon = new ImageIcon(direction + "board\_wPoint.jpg");

ImageIcon whiteBPointIcon = new ImageIcon(direction + "white\_bPoint.jpg");

ImageIcon whiteWPointIcon = new ImageIcon(direction + "white\_wPoint.jpg");

ImageIcon blackBPointIcon = new ImageIcon(direction + "black\_bPoint.jpg");

ImageIcon blackWPointIcon = new ImageIcon(direction + "black\_wPoint.jpg");

ImageIcon rightBPointIcon = new ImageIcon(direction + "rightboard\_bPoint.jpg");

ImageIcon rightWPointIcon = new ImageIcon(direction + "rightboard\_wPoint.jpg");

ImageIcon rightWhiteBPointIcon = new ImageIcon(direction + "rightwhite\_bPoint.jpg");

ImageIcon rightWhiteWPointIcon = new ImageIcon(direction + "rightwhite\_wPoint.jpg");

ImageIcon rightBlackBPointIcon = new ImageIcon(direction + "rightblack\_bPoint.jpg");

ImageIcon rightBlackWPointIcon = new ImageIcon(direction + "rightblack\_wPoint.jpg");

ImageIcon leftBPointIcon = new ImageIcon(direction + "leftboard\_bPoint.jpg");

ImageIcon leftWPointIcon = new ImageIcon(direction + "leftboard\_wPoint.jpg");

ImageIcon leftBlackBPointIcon = new ImageIcon(direction + "leftblack\_bPoint.jpg");

ImageIcon leftBlackWPointIcon = new ImageIcon(direction + "leftblack\_wPoint.jpg");

ImageIcon leftWhiteBPointIcon = new ImageIcon(direction + "leftwhite\_bPoint.jpg");

ImageIcon leftWhiteWPointIcon = new ImageIcon(direction + "leftwhite\_wPoint.jpg");

ImageIcon upBPointIcon = new ImageIcon(direction + "upboard\_bPoint.jpg");

ImageIcon upWPointIcon = new ImageIcon(direction + "upboard\_wPoint.jpg");

ImageIcon upBlackBPointIcon = new ImageIcon(direction + "upblack\_bPoint.jpg");

ImageIcon upBlackWPointIcon = new ImageIcon(direction + "upblack\_wPoint.jpg");

ImageIcon upWhiteBPointIcon = new ImageIcon(direction + "upwhite\_bPoint.jpg");

ImageIcon upWhiteWPointIcon = new ImageIcon(direction + "upwhite\_wPoint.jpg");

ImageIcon downBPointIcon = new ImageIcon(direction + "downboard\_bPoint.jpg");

ImageIcon downWPointIcon = new ImageIcon(direction + "downboard\_wPoint.jpg");

ImageIcon downBlackBPointIcon = new ImageIcon(direction + "downblack\_bPoint.jpg");

ImageIcon downBlackWPointIcon = new ImageIcon(direction + "downblack\_wPoint.jpg");

ImageIcon downWhiteBPointIcon = new ImageIcon(direction + "downwhite\_bPoint.jpg");

ImageIcon downWhiteWPointIcon = new ImageIcon(direction + "downwhite\_wPoint.jpg");

ImageIcon upRightBPointIcon = new ImageIcon(direction + "uprightboard\_bPoint.jpg");

ImageIcon upRightWPointIcon = new ImageIcon(direction + "uprightboard\_wPoint.jpg");

ImageIcon upRightBlackBPointIcon = new ImageIcon(direction + "uprightblack\_bPoint.jpg");

ImageIcon upRightBlackWPointIcon = new ImageIcon(direction + "uprightblack\_wPoint.jpg");

ImageIcon upRightWhiteBPointIcon = new ImageIcon(direction + "uprightwhite\_bPoint.jpg");

ImageIcon upRightWhiteWPointIcon = new ImageIcon(direction + "uprightwhite\_wPoint.jpg");

ImageIcon upLeftBPointIcon = new ImageIcon(direction + "upleftboard\_bPoint.jpg");

ImageIcon upLeftWPointIcon = new ImageIcon(direction + "upleftboard\_wPoint.jpg");

ImageIcon upLeftBlackBPointIcon = new ImageIcon(direction + "upleftblack\_bPoint.jpg");

ImageIcon upLeftBlackWPointIcon = new ImageIcon(direction + "upleftblack\_wPoint.jpg");

ImageIcon upLeftWhiteBPointIcon = new ImageIcon(direction + "upleftwhite\_bPoint.jpg");

ImageIcon upLeftWhiteWPointIcon = new ImageIcon(direction + "upleftwhite\_wPoint.jpg");

ImageIcon downRightBPointIcon = new ImageIcon(direction + "downrightboard\_bPoint.jpg");

ImageIcon downRightWPointIcon = new ImageIcon(direction + "downrightboard\_wPoint.jpg");

ImageIcon downRightBlackBPointIcon = new ImageIcon(direction

+ "downrightblack\_bPoint.jpg");

ImageIcon downRightBlackWPointIcon = new ImageIcon(direction

+ "downrightblack\_wPoint.jpg");

ImageIcon downRightWhiteBPointIcon = new ImageIcon(direction

+ "downrightwhite\_bPoint.jpg");

ImageIcon downRightWhiteWPointIcon = new ImageIcon(direction

+ "downrightwhite\_wPoint.jpg");

ImageIcon downLeftBPointIcon = new ImageIcon(direction + "downleftboard\_bPoint.jpg");

ImageIcon downLeftWPointIcon = new ImageIcon(direction + "downleftboard\_wPoint.jpg");

ImageIcon downLeftBlackBPointIcon = new ImageIcon(direction + "downleftblack\_bPoint.jpg");

ImageIcon downLeftBlackWPointIcon = new ImageIcon(direction + "downleftblack\_wPoint.jpg");

ImageIcon downLeftWhiteBPointIcon = new ImageIcon(direction + "downleftwhite\_bPoint.jpg");

ImageIcon downLeftWhiteWPointIcon = new ImageIcon(direction + "downleftwhite\_wPoint.jpg");

JPanel boardPanel = new JPanel();

JFrame f = new JFrame("Finish");

**Main body**

public GoGameMain() {

setMenu(); // show the menu

initChess(board); //set the digital board

setChessIcon(board); // show the image board

boardInt = initChessInt(boardInt);

addChessIntoPanel(board, boardPanel);

this.add(boardPanel);

this.setTitle("Go 9x9 Game");

this.setResizable(false);

this.setLocation(300, 130);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

this.pack();

this.setVisible(true);

}

**Counting liberties**

private int countLiberty(int a, int b, int userFlag) {

ArrayList stonesVisited = new ArrayList();

int findCount;

int countLiberty=0;

int begin=0, end=0;

int newx,newy;

stonesVisited.add(new Point(a,b));

do

{

findCount=0;

for (int i = begin; i <= end; i++)

{

Point newpoint = (Point) stonesVisited.get(i);

newx=newpoint.x;

newy=newpoint.y;

if(boardInt[newx+1][newy]==-userFlag && !stonesVisited.contains(new Point(newx+1, newy)))

{

stonesVisited.add(new Point(newx+1, newy));

findCount += 1;

}

if(boardInt[newx-1][newy]==-userFlag && !stonesVisited.contains(new Point(newx-1, newy)))

{

stonesVisited.add(new Point(newx-1, newy));

findCount += 1;

}

if(boardInt[newx][newy+1]==-userFlag && !stonesVisited.contains(new Point(newx, newy+1)))

{

stonesVisited.add(new Point(newx, newy+1));

findCount += 1;

}

if(boardInt[newx][newy-1]==-userFlag && !stonesVisited.contains(new Point(newx, newy-1)))

{

stonesVisited.add(new Point(newx, newy-1));

findCount += 1;

}

}

begin = end + 1;

end = end + findCount;

}

while (findCount>0);

for(int count=0;count<stonesVisited.size();count++)

{

Point point= (Point) stonesVisited.get(count);

countLiberty+=liberty[point.x][point.y];

}

return countLiberty;

}

**Capturing Stones**

private ArrayList checkEat(int i, int j, int userFlag) {

int a=i,b=j;

ArrayList stonesDead = new ArrayList();

ArrayList temp = new ArrayList();

temp = null;

int n;

if(boardInt[a+1][b]!=-userFlag && boardInt[a][b+1]!=-userFlag && boardInt[a-1][b]!=-userFlag && boardInt[a][b-1]!=-userFlag)

return stonesDead;

a=i+1;

if(boardInt[a][b]==-userFlag)

{

temp = findEatPart(a, b, userFlag);

if (temp != null)

{

for(n=0; n<temp.size();n++)

stonesDead.add(temp.get(n));

}

}

a=i-1;

if(boardInt[a][b]==-userFlag && !stonesDead.contains(new Point(a, b)))

{

temp = findEatPart(a, b, userFlag);

if (temp != null)

{

for(n=0; n<temp.size();n++)

stonesDead.add(temp.get(n));

}

}

a=i;b=j+1;

if(boardInt[a][b]==-userFlag && !stonesDead.contains(new Point(a, b)))

{

temp = findEatPart(a, b, userFlag);

if (temp != null)

{

for(n=0; n<temp.size();n++)

stonesDead.add(temp.get(n));

}

}

b=j-1;

if(boardInt[a][b]==-userFlag && !stonesDead.contains(new Point(a, b)))

{

temp = findEatPart(a, b, userFlag);

if (temp != null)

{

for(n=0; n<temp.size();n++)

stonesDead.add(temp.get(n));

}

}

return stonesDead;

}

private ArrayList findEatPart(int a, int b, int userFlag) {

ArrayList stonesVisited = new ArrayList();

int findCount;

int countLiberty=0;

int begin=0, end=0;

int newx,newy;

stonesVisited.add(new Point(a,b));

do

{

findCount=0;

for (int i = begin; i <= end; i++)

{

Point newpoint = (Point) stonesVisited.get(i);

newx=newpoint.x;

newy=newpoint.y;

if(boardInt[newx+1][newy]==-userFlag && !stonesVisited.contains(new Point(newx+1, newy)))

{

stonesVisited.add(new Point(newx+1, newy));

findCount += 1;

}

if(boardInt[newx-1][newy]==-userFlag && !stonesVisited.contains(new Point(newx-1, newy)))

{

stonesVisited.add(new Point(newx-1, newy));

findCount += 1;

}

if(boardInt[newx][newy+1]==-userFlag && !stonesVisited.contains(new Point(newx, newy+1)))

{

stonesVisited.add(new Point(newx, newy+1));

findCount += 1;

}

if(boardInt[newx][newy-1]==-userFlag && !stonesVisited.contains(new Point(newx, newy-1)))

{

stonesVisited.add(new Point(newx, newy-1));

findCount += 1;

}

}

begin = end + 1;

end = end + findCount;

}

while (findCount>0);

for(int count=0;count<stonesVisited.size();count++)

{

Point point= (Point) stonesVisited.get(count);

countLiberty+=liberty[point.x][point.y];

}

if(countLiberty<1 && !(stonesVisited.size()==1 && a==stones[steps-1][0] && b==stones[steps-1][1] && isKo))

return stonesVisited;

else

return null;

}

private void eat(int x, int y, ArrayList stonesDead) {

int i,j;

for(int a=0;a<stonesDead.size();a++)

{

Point deadpoint = (Point) stonesDead.get(a);

i = deadpoint.x;

j = deadpoint.y;

if (i == 4 && j == 4)

board[i][j].setIcon(upLeftIcon);

else if (i == 4 && j == board[i].length - 5)

board[i][j].setIcon(upRightIcon);

else if (i == board.length - 5 && j == 4)

board[i][j].setIcon(downLeftIcon);

else if (i == board.length - 5

&& j == board[i].length - 5)

board[i][j].setIcon(downRightIcon);

else if (i == 4)

board[i][j].setIcon(upIcon);

else if (j == 4)

board[i][j].setIcon(leftIcon);

else if (j == board[i].length - 5)

board[i][j].setIcon(rightIcon);

else if (i == board.length - 5)

board[i][j].setIcon(downIcon);

else

board[i][j].setIcon(beginIcon);

boardInt[i][j]=0;

board[i][j].addActionListener(this);

}

}

**Suicidal step**

private boolean checkAvailable(int i, int j, int userFlag) {

ArrayList stonesVisited = new ArrayList();

int findCount;

int countLiberty=0;

int begin=0, end=0;

int newx,newy;

stonesVisited.add(new Point(i,j));

boardInt[i][j]=userFlag;

stoneLiberty();

do

{

findCount=0;

for (int a = begin; a <= end; a++)

{

Point newpoint = (Point) stonesVisited.get(a);

newx=newpoint.x;

newy=newpoint.y;

if(boardInt[newx+1][newy]==userFlag && !stonesVisited.contains(new Point(newx+1, newy)))

{

stonesVisited.add(new Point(newx+1, newy));

findCount += 1;

}

if(boardInt[newx-1][newy]==userFlag && !stonesVisited.contains(new Point(newx-1, newy)))

{

stonesVisited.add(new Point(newx-1, newy));

findCount += 1;

}

if(boardInt[newx][newy+1]==userFlag && !stonesVisited.contains(new Point(newx, newy+1)))

{

stonesVisited.add(new Point(newx, newy+1));

findCount += 1;

}

if(boardInt[newx][newy-1]==userFlag && !stonesVisited.contains(new Point(newx, newy-1)))

{

stonesVisited.add(new Point(newx, newy-1));

findCount += 1;

}

}

begin = end + 1;

end = end + findCount;

}

while (findCount>0);

for(int count=0;count<stonesVisited.size();count++)

{

Point point= (Point) stonesVisited.get(count);

countLiberty+=liberty[point.x][point.y];

}

if(countLiberty!=0)

return true;

else

boardInt[i][j]=0;

return false;

}

**Monte Carlo Method**

private Point monteCarloMethod() {

int x,y;

int n=0,m=0;

boolean suicide=false;

do{

suicide=false;

m=0;

do{

x=(int)(Math.random()\*9+4);

y=(int)(Math.random()\*9+4);

m++;

}while(m<50 && (chessInt[x][y]!=0 || (!checkAvailable(x,y,userFlag) && checkEat(x,y,userFlag).isEmpty())));

if (m<50){

if(x==4 && y ==4){

if(chessInt[4][5]+chessInt[5][5]+chessInt[5][4]==3\*userFlag)

suicide=true;

}else if(x==4 && y==12){

if(chessInt[4][11]+chessInt[5][11]+chessInt[5][12]==3\*userFlag)

suicide=true;

}else if(x==12 && y==4){

if(chessInt[12][5]+chessInt[11][5]+chessInt[11][4]==3\*userFlag)

suicide=true;

}else if(x==12 && y==12){

if(chessInt[12][11]+chessInt[11][11]+chessInt[11][12]==3\*userFlag)

suicide=true;

}else{

if(x==4){

if(chessInt[4][y+1]+chessInt[4][y-1]+chessInt[5][y+1]+chessInt[5][y-1]+chessInt[5][y]==5\*userFlag)

suicide=true;

}else if(x==12){

if(chessInt[12][y+1]+chessInt[12][y-1]+chessInt[11][y+1]+chessInt[11][y-1]+chessInt[11][y]==5\*userFlag)

suicide=true;

}else if(y==4){

if(chessInt[x+1][4]+chessInt[x-1][4]+chessInt[x+1][5]+chessInt[x-1][5]+chessInt[x][5]==5\*userFlag)

suicide=true;

}else if(y==12){

if(chessInt[x+1][12]+chessInt[x-1][12]+chessInt[x+1][11]+chessInt[x-1][11]+chessInt[x][11]==5\*userFlag)

suicide=true;

}else{

if(chessInt[x][y+1]+chessInt[x][y-1]+chessInt[x+1][y]+chessInt[x-1][y]==4\*userFlag &&

(chessInt[x+1][y+1]+chessInt[x+1][y-1]+chessInt[x-1][y+1]==3\*userFlag ||

chessInt[x+1][y+1]+chessInt[x+1][y-1]+chessInt[x-1][y-1]==3\*userFlag ||

chessInt[x+1][y+1]+chessInt[x-1][y+1]+chessInt[x-1][y-1]==3\*userFlag ||

chessInt[x+1][y-1]+chessInt[x-1][y+1]+chessInt[x-1][y-1]==3\*userFlag))

suicide=true;

}

}

if(suicide){

n++;

chessInt[x][y]=0;

}

}else{

return null;

}

if(n>10){

return null;

}

}while(suicide);

return new Point(x, y);

}

**References**

[1] Jay Burmeister and Janet Wiles, ***AI Techniques Used in Computer Go*,** Schools of Information Technology and Psychology, The University of Queensland, Australia

[2] S.C. Hsu, D.Y. Liu, ***The design and construction of the computer Go program Dragon 2***, Computer Go 16 (1991) 3–14.

[3] M. Boon, ***A pattern matcher for Goliath***, Computer Go 13 (1990) 12–23.

[4] MoGo: a Grid5000-based software for the Game of Go, http://www.lri.fr/~teytaud/mogo.html (2012)

[5] EGC 2011 19x19 Computer Go, ***Result***, http://www.gokgs.com/tournList.jsp (2011)

[6] EGC 2011 9x9 Computer Go, ***Result***, http://www.gokgs.com/tournList.jsp (2011)

[7] R. Bozulich, ***The Go Players Almanac***, Ishi Press, Tokyo, (1992).

[8] Wikipedia, ***List of Go terms***, http://en.wikipedia.org/wiki/List\_of\_Go\_terms#Seki(2012)

[9] N. Metropolis and S. Ulam, ***The Monte Carlo Method***, Journal of the American Statistical Association, 44 (1949) 335-341.

[10] Stuart Russell, Peter Norvig, ***Artificial Intelligence a Modern Approach***,p163-171(2010)