TALLINN UNIVERSITY OF TECHNOLOGY
Information Technology
Department of Computer Science
Chair of Network Software

Self-teaching Gomoku player using composite patterns with adaptive scores and the implemented playing framework

Bachelor thesis

Student: Jaroslav Kulikov
Student code: 112662
Supervisor: Ago Luberg

Tallinn
2014

Copyright Declaration

I declare that I have written this Bachelor thesis independently and without the aid of unfair or unauthorized resources. Whenever content was taken directly or indirectly from other sources, this has been indicated and the source referenced. This Bachelor thesis has neither previously been presented for assessment, nor has it been published.

---------- ----------

(*Date*) (*Signature*)

Annotatsioon

Peamiseks lõpputöö eesmärgiks on iseõppiva Gomoku mängija loomine Java keeles. Mängija peab õppima mängima ning kasutama saadud teadmisi käigu valimisel.

Mängija peab tuvastama tundmatuid mustreid käiku valides. Süsteem peab suutma mustreid luua ja defineerida. Gomoku mängija peab tuvastama mustri, mis viitas võidule, et see ära õppida ja järgmiste mängudes kasutada. Loodud süsteem peab töötama piisavalt kiiresti, et käigu valimise algoritm suudaks vaadata palju käike ette.

Mängu lõpus leiab mängija mustri, mis oli võidu põhjuseks, salvestab selle uue mustrina või muudab olemasoleva skoori. Mustrite salvestamiseks kasutatakse selle jaoks välja mõeldud mustrite formaati. See formaat võimaldab hoida sarnased mustrid ühe esitusena andmebaasis, suurendades sellega õppimise kiirust ja efektiivsust. Käigu tegemise ajal otsib mängija andmebaasis olevaid mustreid mängulaualt ja kasutab leitud kombinatsioone võimalike käikude hindamiseks. Efektiivsuse suurendamiseks teatud kasutud käiguvõimalused jäetakse läbi vaatamata.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 30 leheküljel, 11 peatükki, 20 joonist, 2 tabelit ja 4 koodi näidet.

Abstract

The main aim of this thesis is to make adaptive computer Gomoku player using Java. Player also has to learn how to play and make decision where to move using obtained knowledge.

The computer player has to detect undefined patterns, while choosing a move. Another problem is creating and defining patterns. The computer player has to know which pattern has lead to the victory in order to learn it and use in the next games. That algorithm has to work fast enough in order to overview as many moves ahead as possible.

At the end of the game the computer player finds a pattern, which had lead to the victory, saves it or modifies its' score. To save the patterns the computer player uses a pattern format. That format enables to store similar patterns in one, thus increasing learning speed and efficiency. While choosing the move, the computer player scans the game board for pattern-like situations and compares the found patterns with those stored in the database to choose between different move options. To further improve the efficiency of the computer player, useless move positions are not considered.

The thesis is in English and contains 30 pages of text, 11 chapters, 20 figures and 4 code snippets.

List of abbreviations

AI – Artificial Intelligence [1]

MVC – Model-View-Controller [2]

POJO – Plain Old Java Object [3]

GUI – Graphical User Interface [4]

List of figures

List of tables

List of code snippets

Table of contents

1. Introduction

There are many AI players in the world for games like chess, checkers, or k-in-a-row games. One of the issues that developers have to solve while writing an AI player is pattern's setting. The quality of these patterns depends on programmers, their game skills and imagination, which do not tend to be ideal.

The aim of this thesis is to make Gomoku [5] AI player find and rate patterns itself.

1.1. Goals

- Implement a Java application for testing and teaching the Gomoku player.
- Find the patterns, which had lead to the victory.
- Make score for those patterns.
- Evaluate the game board considering patterns found already.
- Make move decision based on evaluated board.

1.2. Methods

In order to achieve the goals board scanning algorithm is needed. Scanning for patterns known already is not enough, because AI player should learn new ones so this algorithm must scan for pattern-like check combinations.

Scoring is achieved by increasing the score of winning templates. Since patterns count in the database might be too big to scan the board for all of them, it is reasonable to use all pattern-like templates searching algorithm, described in the chapter 6. After the algorithm has found all patterns it must search for their score in the database.

Move decision is based on the Minimax [6] algorithm.

2. Gomoku

Gomoku is an abstract strategy board game. Alse called Gobang or Five in a Row, it is traditionally played with Go pieces (black and white stones) on a 19x19 board, however in this thesis board size is custom. Players alternate in placing a stone of their color on an empty intersection or cell (depends on board type). The winner is the first player to get an unbroken row of five stones horisontally, vertically or diagonally.

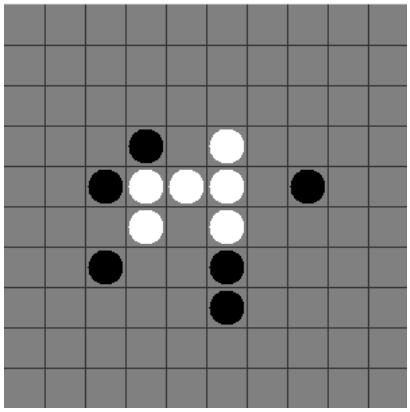

Figure 2-1: Gomoku example

3. System implementation

In order to use and test the AI player the implementation of Gomoku has to be created like the one in [7]. According to the MVC pattern [2] the implementation divides into three main parts: model, view and controller.

3.1. MVC

The main idea of the MVC pattern (Figure 3-1) is to divide rendering, logic and data model. The MVC pattern allows changing things quickly without too much rework of code in all layers of the application. The model doesn't know anything about how to draw itself, or how to change its state. The controller is in charge of changing the models' state and notify the renderer. The renderer has to have a reference to the model and its state, in order to draw it.

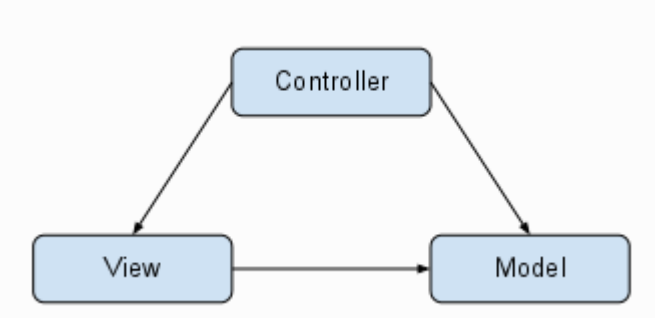

Figure 3-1: Model-View-Controller pattern

3.2. Model

Model classes are usually called POJO, because they are just containers of the information. POJOs don't know anything about how to draw themselves or change their state. The classes in the model package are GameModel, Check, Move, MoveList and MoveNode.

GameModel is the main model class. It holds current board state, memorizes all moves. The game model class also holds the information about a current board size, game rules, winning player and the color to move next. For the sake of convenience the game model class also performs some controller tasks. The game model controls if the game is over after each move. It uses the location of the last move to find the winning five looking around the move position, thus the whole board does not have to be scanned.

Moves are held in the Move class. Move class knows only "x" and "y" coordinates and the color of check used.

MoveList is a container. It holds the last MoveNode, returns the last move, removes and adds moves. MoveNode is a typical list, where each element knows only about itself and a parent node.

3.3. View

GUI is divided into three different classes: user interface, board and main class, which contains two other classes. All those classes are extensions of JFrame [8] elements.

User interface (Figure 3-2) is an extension of the JPanel [9] class. It enables changing the board size, choosing enemy and starting new game. The minimum width and height of the board is five cells and maximum is 100 cells. Different game modes are included. The playing against AI and human players is enabled. The option of observing two AI players' game is also enabled.

Figure 3-2: User Interface

Board (Figure 3-3) class extends the JPanel as well. The board shows a current game state to a user. It is repainted each time the move is made. The board does not really know about how the action listeners work. The board just holds them.

The main GUI class (Figure 3-4) holds two classes mentioned above and redirects commands from the controller to them. It also disposes them in the space and shows the state of the game.

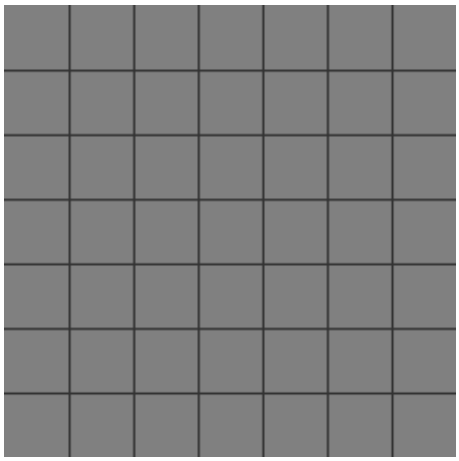

Figure 3-3: The Board

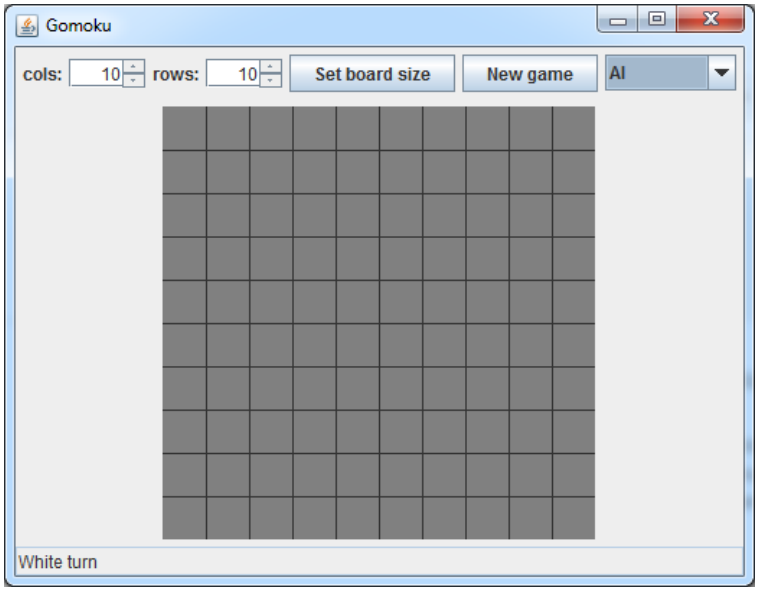

Figure 3-4: All GUI together

3.4. Controller

Both AI and human players as well as the Referee belong to the controller package. AI player is a part of the controller, because it behaves exactly like a human player.

An abstract player class has a link to the Game Model object and contains its' color. It has to be able to make move and has a function which is called when the game is over.

The human player class enables board action listener [10] when the move function is called. On action this class commits the move according to the place clicked. The only thing human player needs to do when the game is over is to notify the user. The human player class controlls all the action listeners. Here is a simple example of how it is done [11].

Code snippet 1: The usage of an action listener

```
public class HumanPlayer extends Player {
        public HumanPlayer(GameModel game, GomokuGUI view) {
            super(game);
            this.view = view;
            view.addNewGameButtonListener(new NewGameButtonListener());
        }
        class NewGameButtonListener implements ActionListener {
```

```
            public void actionPerformed(ActionEvent e) {
                //some code here
            }
        }
}
public class UserInterface extends JPanel {
        public void addNewGameButtonListener(ActionListener newGameButtonListener){
            newGameButton.addActionListener(newGameButtonListener);
        }
}
```

4. Minimax

Minimax [6] is a decision rule used in decision theory, game theory, statistics and philosophy for minimizing the possible loss for a worst case (*max*imum loss) scenario. Alternatively, it can be thought of as *maxi*mizing the *min*imum gain (**maximin** or **MaxMin**). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves.

The algorithm obtains the board state, from where all possible next game states are expanded, and the depth of the search. It builds the tree of all possible move nodes divided into layers with selected depth. Each node gets the score based on its board state in the end of the tree. Each layer of that tree alternates between maximizing levels, where the aim is to benefit player choosing node with a maximum score, and minimizing levels, where the aim is to benefit the opponent by choosing node with a minimum score. Algorithm returns the score of a chosen move. I have upgraded nodes to memorize not only the chosen score, but also the move so it is easier to find the chosen move.

Code snippet 2: Minimax pseudocode

```
function minimax(node, depth, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        bestValue := -∞
        for each child of node
            val := minimax(child, depth - 1, FALSE)
```

```
        bestValue := max(bestValue, val);
    return bestValue
  else
    bestValue := +∞
    for each child of node
        val := minimax(child, depth - 1, TRUE)
        bestValue := min(bestValue, val);
    return bestValue

(* Initial call for maximizing player *)
minimax(origin, depth, TRUE)
```

4.1. Minimax performance optimization

In this thesis the implementation of the minimax algorithm is fastened by an alpha-beta pruning, child game states limitation and sorting by heuristic.

4.1.1. Alpha-beta pruning

The alpha-beta pruning [12] is an addition to the minimax algorithm that decreases the number of nodes that are evaluated, thus the search time can be limited. The main idea is to memorize not only current layer value, but also previous' one. The values of maximizing nodes are called alpha and the values of minimizing nodes are called beta. If occurs, that alpha becomes larger than beta, the maximizing layer will not lower alpha, but the minimizing layer already has a node with a lower value. Or opposite, the minimizing player will not choose a higher value, but the maximizing player already has a bigger value. Then the previous node will not choose this node anyway, so this node is cut off. The alpha and beta values are added to the input of the improved algorithm.

Code snippet 3: Alpha-beta pseudocode

```
function alphabeta(node, depth, α, β, maximizingPlayer)
    if depth = 0 or node is a terminal node
        return the heuristic value of node
    if maximizingPlayer
        for each child of node
```

```
            α := max(α, alphabeta(child, depth - 1, α, β, FALSE))
            if β ≤ α
                break (* β cut-off *)
        return α
    else
        for each child of node
            β := min(β, alphabeta(child, depth - 1, α, β, TRUE))
            if β ≤ α
                break (* α cut-off *)
        return β


(* Initial call *)
alphabeta(origin, depth, -∞, +∞, TRUE)
```

4.1.2. Child game states limitation

The farther an empty board position is from a black or a white check, the less possibly the player will benefit from moving there. So it is reasonable to observe only board positions in the radius of two near board positions with checks on them [13] (Figure 4-1).

To make searching for these positions faster, the parent node search result and the move that belongs to that node are used. If it is the first node, then the available positions are found by brute force search, otherwise the available move positions of the previous node are taken and available positions around the last move are added (Figure 4-2).

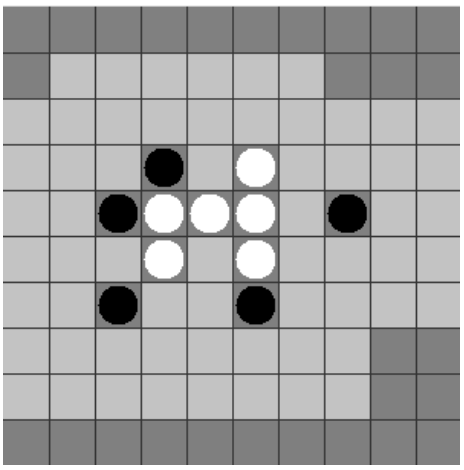

Figure 4-1: Only bright position are observer by the minimax algorithm.

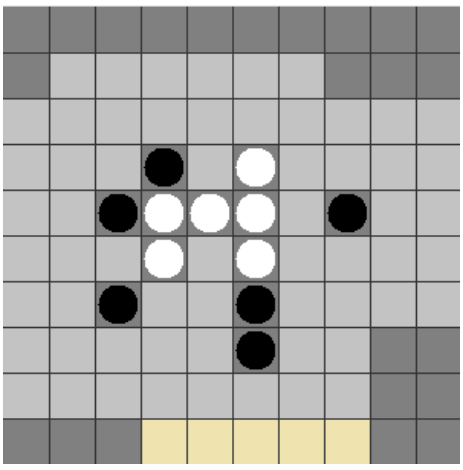

Figure 4-2: With the next move yellow positions are added to the available position list.

4.1.3. Node sorting by heuristic value

When the node is created its board state is pre-evaluated. Only simple linear patterns and their reflections are searched (Figure 4-3). Each pattern has its own score. Patterns are placed in a decreasing sequence, so if one is found, search algorithm is stopped and the node gets its heuristic value equal to the score of the pattern found, because next patterns will have a worse score, otherwise node gets heuristic equal to 0.

Algorithm searches on each horizontal, each vertical and each diagonal. Every new considered position is compared to the patterns' char on the position number equal to counter. If position corresponds with the pattern, the counter is increased by 1, else the pointer moves back by the amount equal to the counter and the algorithm continues searching from the beginning. If the counter reaches the amount of four, pattern is found. Pattern's value is returned.

```
xxxxx = 100
xxxx- = 40
-xxxx = 40
xxx-x = 40
x-xxx = 40
xx-xx = 40
-xxx- = 30
xxx-- = 20
--xxx = 20
```

Figure 4-3: Searched patterns. 'x' means check and '-' means empty

Each node has a Priority Queue [14], which sorts the child nodes by their heuristic value. When minimax algorithm requests the next child of this node, it removes the child node with the best heuristic and returns it.

Heuristic values do not change the minimax results.

5. Pattern format

The pattern is held in a simple String with the maximum size of 14. Patterns may be presented in two ways. The first one is for the simple linear combinations (Figure 5-1). Empty spots will be written as '-' and checks as 'x'.

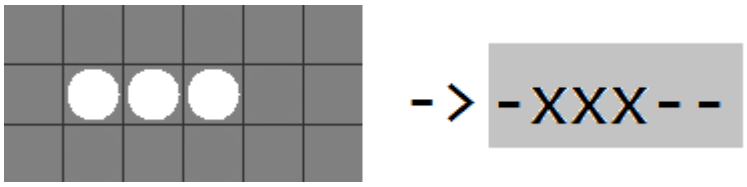

Figure 5-1: Left game state is transformed into right String representation.

The patterns are turned around so that the biggest amount of the checks must be leftwards (Figure 5-2).

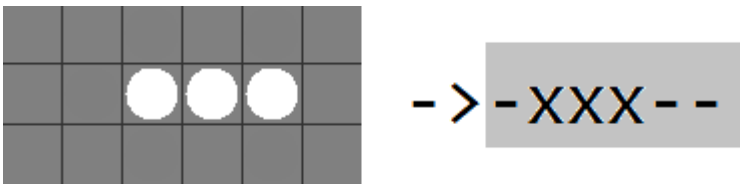

Figure 5-2: Pattern if turned around.

Quite ofter the victory is achieved by using the double threats [15]. One is used to make the opponent defend himself spending his move to block the threat, while another threat stays open (Figure 5-3). For that purpose the composite pattern standard is used.

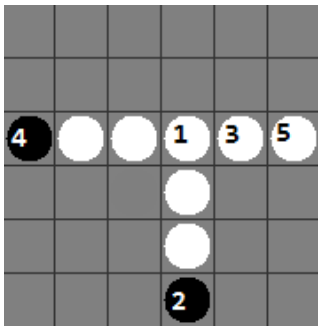

Figure 5-3: Composite pattern usage.

Composite patterns are also written in one line (Figure 5-4). The first part is the main linear pattern. The last part is the secondary linear pattern. Between them two numbers are placed. The first one is the position where the main pattern is crossed by the secondary and the second one is the position where the secondary pattern is crossed.

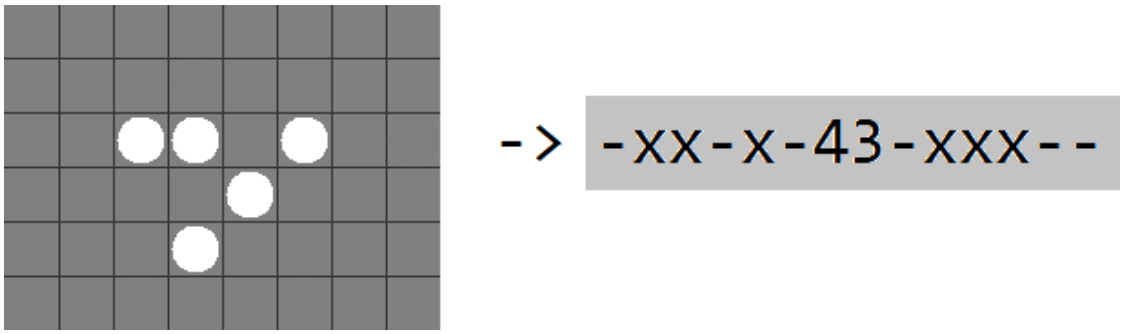

Figure 5-4: Composite pattern example.

The advantage of this format is an independence from direction. There is no need to turn the template in order to find it on the board. Each part of the pattern can be found on any direction,

thus the amount of total patterns will be reduced. Searching for a double-threat reduces the load of the Minimax algorithm. The winning combination can be found with a less searching depth. All combinations showed below will have the same String representation (Figure 5-5).

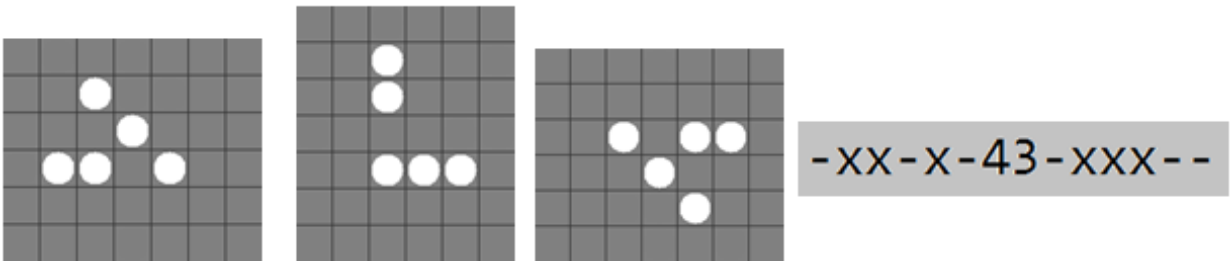

Figure 5-5: Different representations of single String pattern

The disadvantage of this format is limitation by only two linear combinations per pattern. The third combination may be used to make the opponent move somewhere else and let the player complete the winning combination (figure 5-6). The bottom three makes the opponent close it in fear of the four, which is opened from the both sides ("-xxxx-"), thus enabling the player make two opened threes with the next move, thus the player will be able to complete the "-xxxx-" pattern anyway. The usage of the Minimax compensates this disadvantage observing multiple moves further, how can double or even single threat be created.

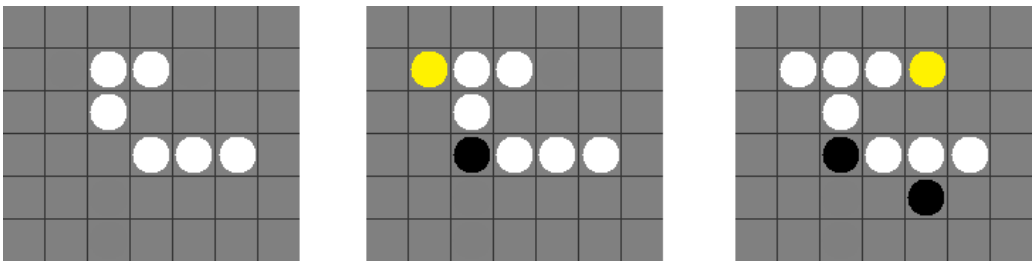

Figure 5-6: Usage of triple pattern

6. Game state evaluation

An evaluation algorithm scans the board horizontally, vertically and diagonally for all pattern-like templates. The algorithm knows if a white check and a black check occurred in the last five board positions. If the algorithm found a combination of five positions in a row, where there are at least two checks of the one color and zero of another, then new pattern is created and saved.

Code snippet 4: Finding linear pattern pseudocode

```
function findLinearTemplate(x, y, direction){
```

```
if(white checks expire in > 0){
    white checks expire in--;
    if(white checks expire in == 0 ){
        white check = false;
    }
}
If(black checks expire in > 0){
    black checks expire in--;
    if(black checks expire in == 0 ){
        black check = false;
    }
}
if(board[x][y] == white check){
    white check = true;
    white checks expire in = 5;
} else if(board[x][y] == black check){
    black check = true;
    black checks expire in = 5;
}
if(the board doesn't end in at least 4 positions behind){
    if((white check && !black check) || (!white check && black check)){
        Pattern p;
        if(there are at least 2 checks)
            create new pattern;
            add pattern;
        }
    }
}
```

The main problem of this algorithm is finding the excessive patterns. For example, if there is an occurrence of three in a row with enough empty space around, the algorithm will create five different patterns (Figure 6-1). Because of that, the overlapping patterns should be removed, but only those, where the amount of

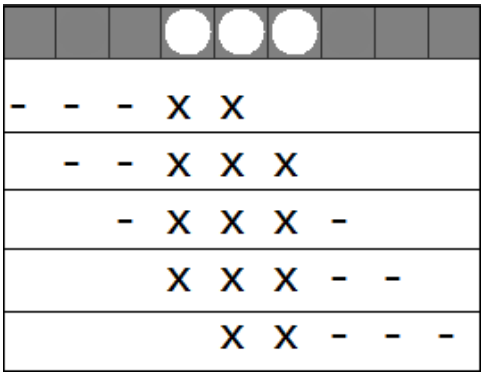

Figure 6-1: Five patterns from one

checks is less than in the other. Deleting the overlapping patterns with the same amount of checks is not necessary, because each of them might be crossed over by another pattern in the position, that another does not cover, so some valuable patterns might be lost. In the case showed in the figure 6-1 only the first and fifth patterns should be removed.

The next step of the evaluating algorithm is to search for pattern intersections. If an intersection is found, the composite pattern is created.

During the execution of the algorithm many equal patterns might be found, so they must be deleted. This can't be done before the intersections are found, because linear patterns are linked with the positions they are found in, so if one is deleted, its intersections will be lost.

After all the steps above are completed, all found patterns are searched in the database. All scores of the patterns found are accumulated. The score of patterns, which belong to the opponent, are subtracted. The sum is returned.

6.1. Evaluation algorithm performance optimization

Connecting to the database is a time consuming process. Considering that the board is evaluated quite often, it is a wasting of a precious time. The database is not changed until the end of the game, so it is reasonable to save all patters in the memory only once, when the game starts.

Patterns should be found quickly, so they are held in the HashMap [16] where String pattern is the key and score is the value.

6.2. Pattern extension

Sometimes the sixth position of the linear pattern is needed. The best example is four checks with an empty slot at each edge ("-xxxx-"). The strength of that pattern is the ability to put the last check at any edge. So it doesn't matter where the opponent will move, the player will be able to finish his winning five.

Not every pattern is required to be extended. Only the opposite side of an empty slot is expanded. If both edges have an empty slot, two different extensions are made. There is no need to expand pattern, which is "blocked" from each side, because it is fixed on its place (Figure 6-2).

```
XXX--  ->  -XXX--
X-XXX  ->  X-XXX
```

Figure 6-2: Pattern extension example

7. Finding the winning patterns

The AI player has a Teacher class. The Teacher gets a final game state and the move list in the end of the game. The task of the Teacher class is to find the winning five, identify if it has a useful crossing pattern, and score the pattern found.

The last move is used to find the winning five more quickly. The algorithm finds five-in-a-row location and starts returning to the previous game states using the move list. Each algorithm step the Teacher removes two last moves, the one move of each color, updates the winning pattern and searches for the crossing pattern if it wasn't found before. Both winning and crossing patterns are also expanded, if it is possible and necessary, as it is described in the previous chapter. The algorithm continues working until the check amount in the winning pattern is bigger than one. Each time the score of the found pattern is modified. It doesn't matter if the AI player has lost or won. The Teacher finds and scores a winning pattern independently of the color.

7.1. Searching for the crossing pattern

The area of our interest locates in the radius equal to four around each winning five position. The search is carried by the function, that gets the position to look around and the direction, thus the function is called three times for each position of the winning five. For example, if the winning five is located vertically, then we should search on the horizontal and two diagonals.

Patterns of each direction are searched separately. The searching function gets position coordinates to look around and direction of the search. The search process is similar to the algorithm described in the chapter 6. The difference is in the search area. Only the area in radius of 4 is scanned. The template with the best amount of the checks is chosen among all patterns found.

7.2. Crossing pattern acceptance

Not every crossing pattern is acceptable. Pattern can't influence the game result if it does not contain enough checks. The first condition is to contain two or more checks. Crossing pattern also has to be long enough in order to be as valuable as the winning pattern, in the current game state. It has to contain as many checks as the winning pattern has. If the crossing pattern is not valuable enough, it can't influence the game.

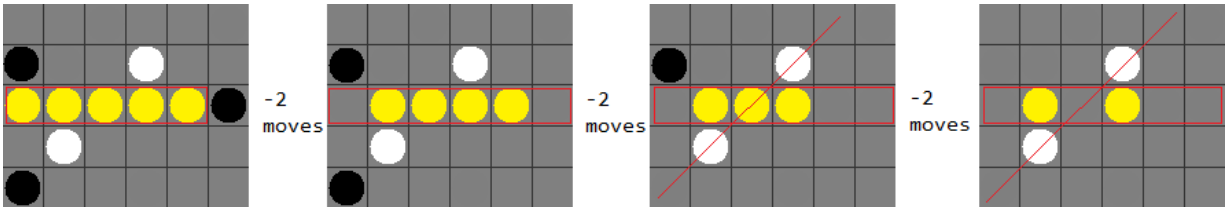

Figure 7-1: Winning checks – circled and crossing checks – line through.

7.3. Making score

Each found pattern is converted in to String according to previously, described format in the chapter 5. The pattern Converted in to String is searched in the database. This pattern is added to the database, if there was no such pattern yet, else the score of the found pattern is increased by three.

The possible score of new patterns is divided into three layers. Each score layer is ten times bigger than the previous. Each layer has a range of possible score values: one hundred below the default and one hundred above. The default values of each layer are 100, 1000 and 10000. The value of 100000 is used to identify the victory. The default value of each pattern is calculated using a formula *"10^checks_in_the_winning_pattern"*, if the pattern is linear. In case of the composite pattern is used the formula *"10^(checks_in_the_winning_pattern+1)"*. The pattern with four checks is an exception. Even being composite, the second formula cannot be used, because otherwise the default value of those patterns will reach the winning value, which is unacceptable.

The layered score system is used in order to enable summing up all found pattern scores while board state evaluating. Thus the sum will hardly reach the winning score. Another advantage is clear difference between patterns with different amount of checks. The pattern with only two checks will never reach the one with three.

It is hardly possible to guess what pattern has led the player to the loss. Score of the patterns is decreased, if AI player has lost. That means, that the patterns he owns are not as good as needed.

As future work the score giving and changing system should be changed. In this thesis only the amount of checks is considered, while determining a score layer, but check position is also valuable.

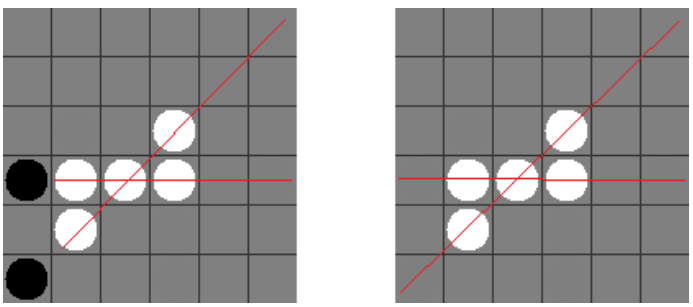

Figure 7-2: The second pattern is more valuable, because is can be extended from the both edges, but these patterns will get the same scores.

8. Results

id	template	score
150	-xxxx-	10006
151	-xxx--	999
152	-xxx--12--x-x-	9996
153	--xx--	96
154	xxxx-	9996
155	--xx--32xxxx-	996
156	--xx--32xxx--	996
157	-xx-x-	998
158	--x-x-23-x-x--	998
159	-xxx--33-xxx--	10000
160	-xx---33-xx---	1000
161	-xx---	100

Table 8-1: 5 games

id	template	score
150	-xxxx-	10100
151	-xxx--	1014
152	-xxx--12--x-x-	9938
153	--xx--	38
154	xxxx-	9959
155	--xx--32xxxx-	938
156	--xx--32xxx--	938
157	-xx-x-	959
158	--x-x-23-x-x--	940
159	-xxx--33-xxx--	9947
160	-xx---33-xx---	947
161	-xx---	52
162	--x-x-40x---x	942
163	--x-x-23-xxx--	942
164	-xx-x-43xxxx-	9942
165	-xx---43xxx--	942
166	-xx---43x-x--	942
167	-xxx--12xxx--	9942
168	--xx--42xx---	942
169	xxx-x	9944
170	xxx--	944
171	xxx--01xx---	9944
172	xx---01xx---	944
173	-x-x--	44
174	-xx---21xxx--	952
175	-xx---10xx---	957
176	--xx--20xx---	951
177	xxx--12-xxx--	9953
178	--x-x-41-xx---	953
179	--xx--30xx---	950

Table 8-2: 30 games

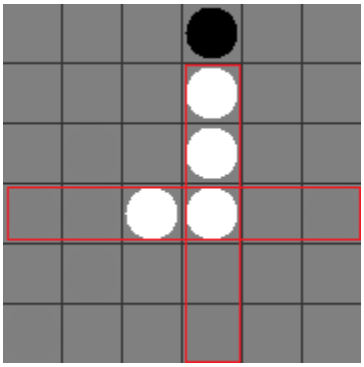

Figure 8-1: Pattern id 156

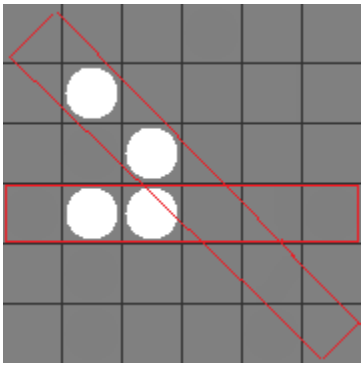

Figure 8-2: Pattern id 160

In the tables 8-1 and 8.2 the results of self-teaching after several games are shown. The score of the first pattern with id 150 has risen to its maximum value, because it is used in almost every game. The other patterns are not used often, so their scores do not rise quickly. As supposed, all patterns are clearly divided into layers by their score. New patterns are added seldom, because

many patterns are already covered. Almost every pattern score has decreased because of the system used to decrease pattern scores. If the pattern is not used in a lost game, then its score is decreased. For that reason I can state, that score system needs to be improved.

9. Conclusion

The main aim of this thesis was to create an AI Gomoku player, which is able to learn, rate and use patterns itself. Furthermore, the implementation of the Gomoku game was needed in order to test the AI player.

A pattern format was created. Using this format the amount of patterns used can be reduced, thus the AI player learns faster. The AI player is not only able to search predefined patterns but also can look for potential patterns. Usage of this search reduces the amount of board scanning, because in case of predefined pattern searching the board should be scanned newly for the each pattern, while all potential patterns can be found by only one board scanning. The board evaluation is made using the following algorithm, described in the chapter 6. The score of the board state is calculated using found patterns. Pattern recognition also allows learning new patterns. Furthermore, patterns are not only found, but also scored according to their win rate and their "distance" to the victory. The implementation of the Minimax algorithm is used to choose the move. The Minimax algorithm is speeded up enough to search in the depth of four. The convenient interface is built for playing, testing, and training. Due to the MCV pattern each part of the program can be changed quickly and almost independently.

According to these results I can state that AI player really benefits from self-teaching. It increases the range of predefined patterns and self-teaching does not slow down the game process.

As an extension, more complicated and effective scoring system can be implemented. For example the tournament between AI players, every one of which uses only one pattern from the database. Each win will gain score for pattern used by a winner and loss will reduce the score. Draw will also decrease the score, but less.

10. References

[1] "Artificial intelligence," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Artificial_intelligence. [Accessed 8 June 2014].

[2] "Building Games Using the MVC Pattern – Tutorial and Introduction," Obviam, 5 February 2012. [Online]. Available: http://obviam.net/index.php/the-mvc-pattern-tutorial-building-games/. [Accessed 29 May 2014].

[3] "Plain Old Java Object," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Plain_Old_Java_Object. [Accessed 8 June 2014].

[4] "Graphical user interface," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Graphical_user_interface. [Accessed 8 June 2014].

[5] "Gomoku," Wikipedia, 15 May 2014. [Online]. Available: http://en.wikipedia.org/wiki/Gomoku. [Accessed 28 May 2014].

[6] "Minimax," Wikipedia, 26 April 2014. [Online]. Available: http://en.wikipedia.org/wiki/Minimax. [Accessed 28 May 2014].

[7] F. Swartz, "Gomoku implementation," 20 November 2005. [Online]. Available: http://www.leepoint.net/notes-java/examples/games/five/five.html. [Accessed 14 April 2014].

[8] "JFrame," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/javax/swing/JFrame.html. [Accessed 6 June 2014].

[9] "JPanel," Oralce, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/javax/swing/JPanel.html. [Accessed 6 June 2014].

[10] "ActionListener," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/awt/event/ActionListener.html. [Accessed 29 May 2014].

[11] F. Swartz, "Model-View-Controller (MVC) Structure," 2004. [Online]. Available: http://www.leepoint.net/notes-java/GUI/structure/40mvc.html. [Accessed 6 June 2014].

[12] "Alpha-beta pruning," Wikipedia, 29 May 2014. [Online]. Available: http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. [Accessed 30 May 2014].

[13] A. Loos, "Machine Learning for k-in-a-row Type Games," 2012. [Online]. Available:

http://dspace.utlib.ee/dspace/bitstream/handle/10062/32992/thesis.pdf?sequence=1. [Accessed 29 May 2014].

[14] "PriorityQueue," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html. [Accessed 29 May 2014].

[15] "stackoverlfow," 8 August 2011. [Online]. Available: http://stackoverflow.com/questions/6952607/ai-strategy-for-gomoku-a-variation-of-tic-tac-toe. [Accessed 5 June 2014].

[16] "HashMap," Oracle, [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html. [Accessed 29 May 2014].

11. Extra

The source code can be downloaded using this link:

http://dijkstra.cs.ttu.ee/~t112662/bsc/SelfTeachingGomoku.rar.