Knowledge Based Systems
Yavalath

Names:	Schabert Christoph, Alaniz Stephan, Brodhacker Jan
Course:	TINF-AI11/BC
Lecturer:	Hr. Dr. Vossen
working period:	09.09.2013 - 14.1.2014

Statutory declaration

I herewith declare that I have completed the present thesis independently making use only of the specified literature and aids. Sentences or parts of sentences quoted literally are marked as quotations; identification of other references with regard to the statement and scope of the work is quoted. The thesis in this form or in any other form has not been submitted to an examination body and has not been published.

Stephan Alaniz:
Heidelberg, the __.__.2013 ....
(signature)

Christoph Schabert:
Heidelberg, the __.__.2013 ....
(signature)

Jan Brodhacker:
Heidelberg, the __.__.2013 ....
(signature)

Contents

List of Figures

Bibliography

[1] Morten Pahle Arno Hollosi. UCT for Monte Carlo computer go. Online; accessed 28-October-2013.

[2] BoardGameGeek. *Yavalath: On Evolutionary Game Design*, Jun 2011.

[3] Cameron Browne. Yavalath. *www.cameronius.com/games/yavalath/*, 2007.

[4] Cameron Browne. *Evolutionary Game Design*. Springer London, 2011. ISBN: 978-1-4471-2178-7.

[5] Computational Creativity Group, Imperial College London. *Monte Carlo Tree Search*.

[6] GUILLAUME M.J-B. CHASLOT, MARK H.M. WINANDS, H. JAAP VAN DEN HERIK, JOS W.H.M. UITERWIJK, BRUNO BOUZY. Progressive strategies for monte-carlo tree search. *http://www.personeel.unimaas.nl/g-chaslot/papers/newMath.pdf*, 2008.

[7] Jan Schaefer. The uct algorithm applied to games with imperfect information. Master's thesis, Otto-von-Guericke-Universitaet Magdeburg, 2008. http://www.b0n541.net/study/jan_schaefer_uct_algorithm_applied_to_games_with_imperfect_information.pdf.

[8] PETER AUER, NICOLO CESA-BIANCHI, PAUL FISCHER. Finite-time analysis of the multiarmed bandit problem. *http://moodle.technion.ac.il/pluginfile.php/192340/mod_resource/content/0/UCB.pdf*, 2002.

[9] Wikipedia. Monte-Carlo tree search. http://en.wikipedia.org/wiki/Monte-Carlo_tree_search. Online; accessed 28-October-2013.

Chapter 1

Introduction

1.1 Goal and Motivation

The goal of this project is to implement a game called Yavalath and to develop two different artificial intelligences, which are able to play the game and win against nearly every human player.

1.2 Technical Notes

All sources to illustration are listed in the illustrationregister. All illustrations that don't have an explicitly mentioned source are self-created by the authors.

The implementation of the game and the AI-players with a fully functional graphical user interface is availible on GitHub under the following url:

https://github.com/belafarinrod91/KBSYavalath (hep: 25.10.2013)

This project is under no license or otherwise protected, feel free to clone the Git repository and compile it yourself.

This publication was written in LaTeX and compiled with MikeTex.

Chapter 2

The Game - Yavalath

This chapter will explain the creation and history of Yavalath and the game itself.

2.1 History

Yavalath is the most successful game evolved from LUDI in 2007 and published in 2009 by Nestorgames (Spain). LUDI is a game-generating computer program that implements an algorithm for generating new games. This LUDI software system not only defines the scope and rules of the game, it also interpets the game and coordinates testplays to measure the game quality. LUDI was developed by Cameron Browne as part of his PhD in AI and game design. He was also awarded with the Dean's Award for Outstanding Thesis [4].

In 2011 Yavalath was ranked place 99 by the BoardGameGeek community[1] of 4300 abstract games ever invented. It was also ranked place 8 of 200 abstract games invented in 2007.

Yavalath won 2012 the Humies Award for Human-Competitive Result Produced by Genetic and Evolutionarty Computation

2.2 Categorisation

Yavalath is a blend of connect- and patter-building games. The BoardGameGeek community categorised Yavalath as an abstract game because it was the first computer-generated game with such a huge success.

Yavalath also inspired a new game category that's called:

> winning with a line of N and losing with a line of N-1.

The name Yavalath was randomly created by a Markovian algorithm.

[1] `www.boardgamegeek.com`

2.3 The Rules

The rules for Yavalath are pretty simple what makes it easy to learn. But its still novel so that the game is still intressting. One of the reasons why the game became so successful is that Yavalath achieves a good balance between simplicity and innovation.

Yavalath is for two or three players and the average playingtime per game is 10 minutes[2].

The board is a hexagon with 5 spaces on each side, that is initally empty. With these specifications there are 61 free cells in total as shown in the illustration 2.1.

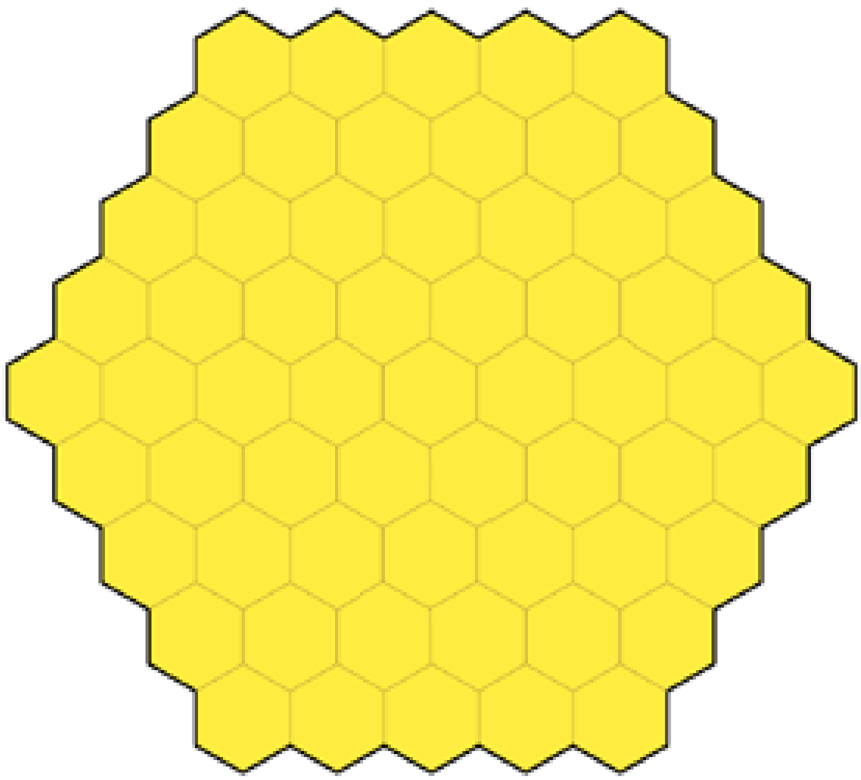

Figure 2.1: The Yavalath game board

Every turn a player places a piece of their colour onto the board.
This goes on until:

- one player has won

[2]this was voted by the BoardGameGeek comunity, source:`http://boardgamegeek.com/boardgame/33767/yavalath`

- all but one player have lost
- all cells are filled and it is a draw

As soon as a player forms a line of three pieces in a row the player loses.
A player can win by two different methods:

1. one player manages to make a line of four or more pieces of his color (In illustration 2.2 the white player won)
2. one player forces another player to form a line of three pieces, which is called a forced move (In illustration 2.3 the black player is forced to move to prevent the white player from winning)

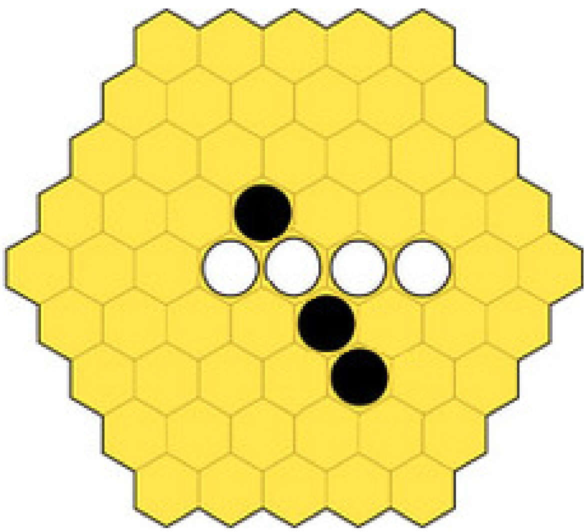

Figure 2.2: Four in a row wining

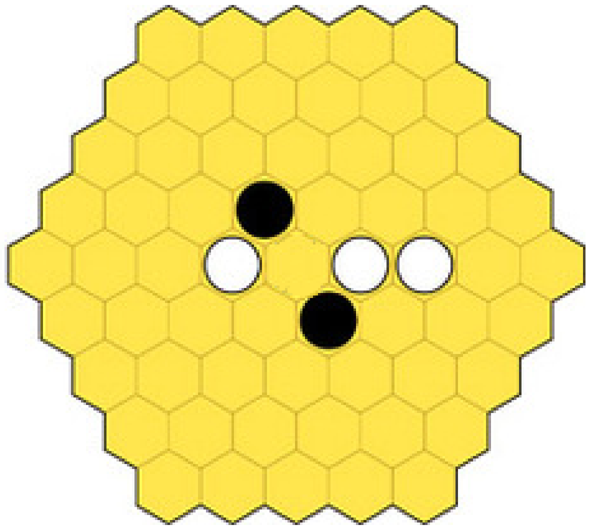

Figure 2.3: Forced move winning

This leads to a so called Rule Tension, in this case four in a row is good but three in a row is bad. This means the player can't just extend lines and must think about the pros and cons of every move he makes.[3]

2.4 Tactics and Strategy

Cameron Browne himself defined some tactics and strategies and described them in his publication. He started to describe a number of choices for the first move and a number of patterns that have a good chance of winning.

The first move is crucial for the following tactics and for the chance to win. Illustration 2.4 shows a Yavalath board with the win chances for the first move, a bigger bubble means a bigger win chance. It visualizes that a centered first move provides the biggest chances of winning.[3]

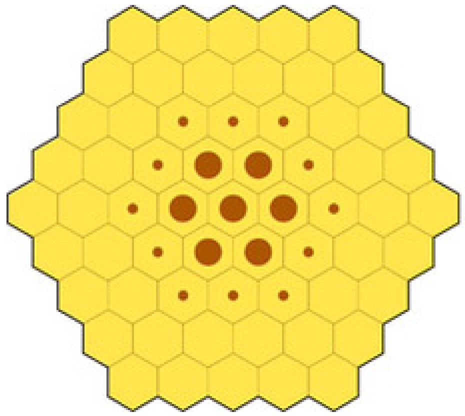

Figure 2.4: The win chance for the first Move

Another good choice for a first move is to start with a pattern. The triangle starting pattern is an excellent example for a first move pattern (the triangle starting pattern is shown in illustration 2.5 on the left side) because it provides a huge number of chances to win by a forced move. Additionally, it's hard to stop a player that used the triangle pattern as a starting move. The only way to stop it is to forsee it pretty early and prevent the player from forming the triangle. The right side of illustration 2.5 shows a possible outcome of the triangle starting position where the black player has lost by a forced move.[3]

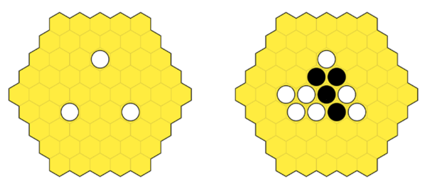

Figure 2.5: The triangle starting pattern

In Yavalath a triangle is always a good pattern like the small triangle (as seen in illustration 2.6). The Reason that the small triangle is such a strong pattern is because it can form a huge number of winning chances.

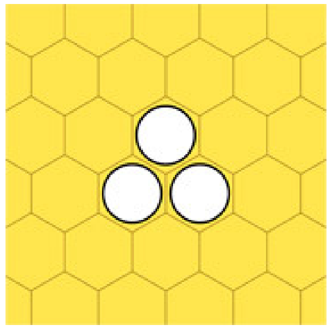

Figure 2.6: A small triangle pattern

To show that the small triangle is a good pattern a winning pattern is shown in illustration 2.7. This forces the black player to lose by a force move. It's also possible to turn a pattern by 120 degrees. With that in mind there are at least three possible winning situations with this simple pattern. If an opponent player tries to form a triangle it is recommended to stop him as soon as possible.[2]

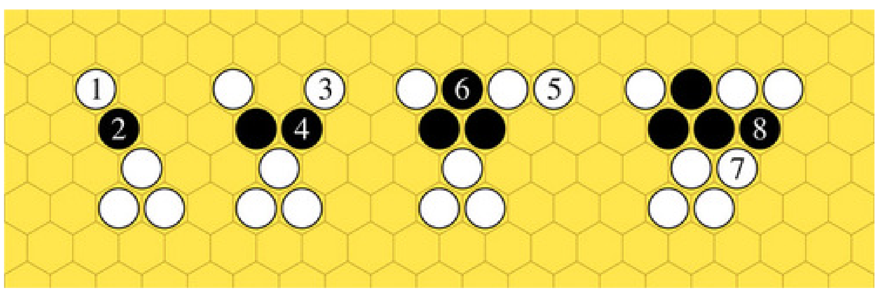

Figure 2.7: A small triangle winning pattern

Chapter 3

Artificial Intelligences

The following chapter describes the two approaches of Artificial Intelligences, which were implemented to this Game. The two Artificial Intelligences are able to play against a human player and even to win against him. On the one hand there is the so called UTCAI, which can has three different levels of difficulty. And the other hand there is the simple approach of a random AI. Both approaches are explained in a more detailed view in the next two sections.

3.1 Random-AI

The random AI is one of the two implemented Artificial-Intelligences, as already mentioned in the description of this chapter. Actually the random AI is defintely the more easier opponent for a human player and also for the other other kind of AI, the UCT AI. Additionally to this fact, the random AI is also the more easier approach to implement. Rather the random AI can be described within to sentences : Set your tokens randomly, but before you do that, check if you could win with your next turn or if you must prevent, that one of your opponents could win. In a more detailed view this means, the random AI sets its token completely randomly to the playground. This happens with the assistance of a special algorithm, which receives the system time. Based on this system time the algorithm can determine a special courton on the playground. If this courton is not used by another player, the random AI will set its token there. It doesn't matter if this turn was smart or if it could yield an advantage.

But before the random AI sets its token, it checks whether a opponent could win on the next turn. This happens based on a predefined rule set, as already mentioned in the chapters above. So figure 3.1 shows a situation, where the red player (random AI) prevents the win of player green, by setting the red token with the white dot. Additionally to that the random AI checks also if it could excel an opponent on the next turn. Figure 3.2 shows how the red player could set a token (red token with white dot) to force player green to loose. For this situation the random AI had to set the other red tokens before. And all good

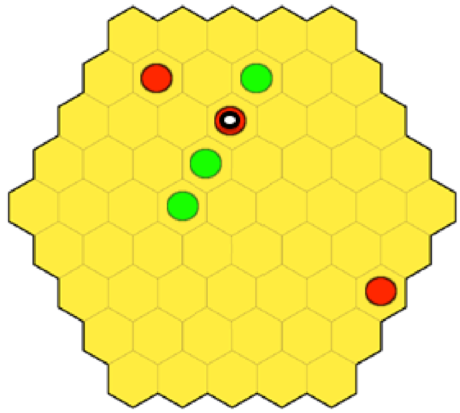

Figure 3.1: Random AI prevents red player from winning

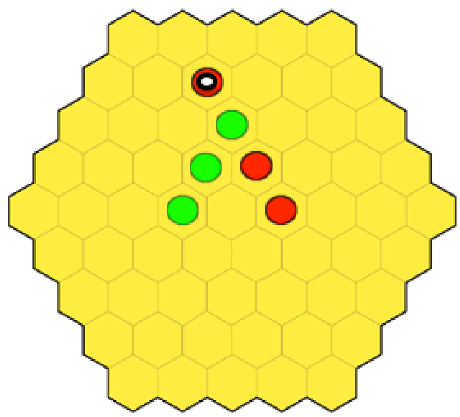

Figure 3.2: Random AI excels player green

things are three, so the random AI checks in a third step if it could win itself, when it sets the token on a certain courton. Figure 3.3 shows that the random AI would recognize the chance to win in this situation. The random AI must only set the red token with the white dot to win. All in all the random AI is not very smart, because neither it can deal with specific game events nor learning from faults, which were made by the random AI in a game before. It actually sets its token so long as the AI itself could win randomly. And this probability is not very high. All in all the random AI is not only a random AI as described before, but it has implemented some trivial logic to recognize if the random AI could win or prevent a win of an other player.

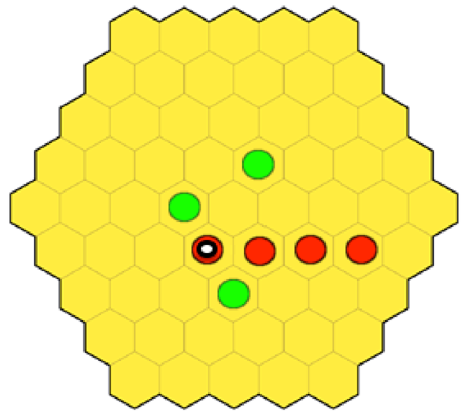

Figure 3.3: Random AI recognizes chance to win

3.2 UCT-AI

The following chapter describes the fundamentals of the UCT (Upper Confidence Bounds applied to Trees) algorithm. It was implemented in order to provide the Yavalath game with an artificial intelligence (AI) to play against. The main task of a game playing AI is to choose its next move with the information about the state of the game. The motivation and reasons behind choosing the UCT algorithm are briefly pointed out. Afterwards it is explained how this algorithms works and how it is integrated into Yavalath.

3.2.1 Why UCT?

The UCT algorithm was developed for decision making problems especially in terms of game playing. It has its strengths in games where:

- the state of a game is hard to evaluatate and
- the game has a high branching factor.[9]

In Yavalath, these two attributes can be found. Firstly, Yavalath has no developed theory, mainly because it is relatively unknown, and therefore there does not exist an evaluation function which could rate the current state of the game. If it was possible to assign a score depending on how good a player's tokens are placed on the board, it would be possible to deploy algorithms (such as Alpha-beta pruning and Minimax), which can leverage this information in order to find a good or even the best possible next move.

Secondly, Yavalath also has a high branching factor. Looking only at the first 10 moves (5 from each player when 2 are playing), there already are $\frac{61!}{51!} = 3,27 * 10^1 7$ different move combinations. Good game playing AIs cannot only look at the next move, but rather need to look ahead a couple of moves.

Otherwise, they could not detect potential threats or opportunities which might lie ahead a few moves. In order to investigate all these branches, the AI needs to be capable of handling the huge number of combinational moves.

The UCT algorithm is able to handle both these attributes of Yavalath and therefore is a reasonable choice for a AI algorithm. This is further justified by the fact that the creator of Yavalath himself implemented this algorithm as AI in his Yavalath application.[3]

3.2.2 How the UCT algorithm works

The UCT algorithm is based on the Monte Carlo Tree Search (MCTS). The MCTS is a best-first search algorithm where the results from previous iterations guide the algorithm to the best path through the tree. It consists of 4 steps which are run in a loop. The 4 steps[5][9] are:

- *Selection:* Starting at root node R, successively select optimal child nodes until a leaf node L is reached.
- *Expansion:* If L is a not a terminal node, which ends the game, then create one or more child nodes and select one C.
- *Simulation:* Run a simulated playout (usually a random playout) from C until a result is achieved.
- *Backpropagation:* Update the current move sequence from C to R with the simulation result.

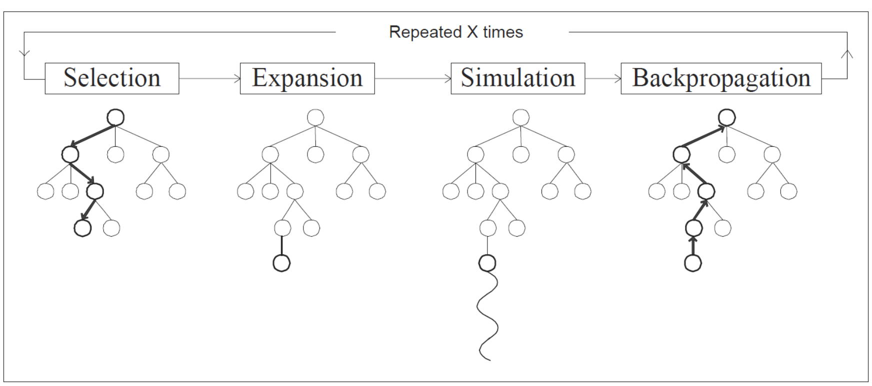

Figure 3.4: Monte Carlo Tree Search

In the regard of Yavalath, the root of the tree would represent the current state of the game. From there the first layer holds all possible next moves the AI can make. The goal of the AI is now to find the best promising move in that first layer. Going down one layer, the second layer would represent all possible counter moves of the following player. In each layer the algorithm tries to find the best move and expands successive moves to see how the game would evolve when picking a particular move in the first place. That said, the algorithm also takes into consideration what the best move for the AI's opponent is. For example, even though a move seems to be promising in the first layer, the opponent might be able to counter this move with a very strong response move and thus leading to a worse situation than initially thought. Therefore, it is important to further investigate potentially good moves by going down more and more layers, i.e. rounds of the game. The simulation phase is used to determine the performance of the moves. Usually, random playouts are used to quickly see how a game of chosen moves turns out in the end. It might seem that a random playout cannot be representative, but with more and more playouts the accuracy of the expected win chance increases due to the law of large numbers. The result of a random playout is backpropagated to all involved moves till the root node.

It is still unclear how the algorithm chooses a node, and path respectively, in the selection phase. This is probably the most important part of the algorithm because it determines the best-first search strategy. The algorithm needs to find a good balance between exploitation and exploration. Expoitation is defined as investigating deep variants of the best move sequences whereas exploration means trying out moves which have not been simulated many times. The former is important because the AI needs to be confident that the choice of a move is good even in the further playout and the latter has to be carried out so that the AI does not miss out on good moves which might just have unluckily performed bad in a few playouts.

To balance out exploitation and exploration the UCB1 (Upper Confidence Bound 1) policy is used which was first introduced by Peter Auer.[8] Before using UCB1 in order to decide which node to select from a set of nodes, every node has to be played out at least once. Afterwards the ith node is chosen which maximizes the equation 3.1.

$$\bar{x}_i + C * \sqrt{\frac{\ln n}{n_i}} \tag{3.1}$$

x_i is the performance measure from node i. The performance measure must lie in the interval [0;1]. For games it usually is the win chance $\frac{w_i}{n_i}$ where w_i is the number of wins from random playouts of node i and n_i is the number of times node i has already been played. n is the overall number of playouts and C is a constant which has to be adjusted empirically for a given problem.[7]

In the UCB1 policy, the first part of the equation x_i stands for the exploitation part. The higher the win chance of a move, the higher the value of the equation. The rest of the equation stands for the exploration part. Assuming a node is visited once and not any more for a couple of times, i.e. n_i is fixed, then only n increases which makes the overall second part of the equation grow. On the contrary, for a node which is often visited n_i is increased simultaneously. As a result, the second part of the equation decreases because the linear function in the denominator rises faster than the natural logarithm in the numerator. This means that nodes with a bad win chance and few plays will eventually be explored too. To what degree and how often they are explored determines the constant C. The higher the value the more exploration is done and the lower the value the more exploitation is done. Auer suggested to set the constant C to $\sqrt{2}$.[8] Therefore, this can be used as a reasonable starting value.

From the UCB1 policy it can be derived that each node needs to to hold two pieces of information: the number of wins and the number of visits. Both the MCTS and the UCB1 policy form the UCT algorithm. This explains where the name Upper Confidence Bounds applied to Trees comes from. The UCT algorithm could be run quite long on highly branching trees. Fortunately, it can be stopped at any time. Of course, the longer the algorithm runs the more statistically significant the result will be, but often a fixed number of iterations (depending on the problem) are sufficient to get a good result. The best move will finally be chosen by looking at the information of all nodes in the first layer. It is best to choose the node which was visited the most. Even though it is possible to choose the node with the highest win chance, this might result in a bad move. Through exploration a node can temporarily have a higher win chance, but as it is probably not exploited very much, it is unclear if it really is a good move. A high win chance will subsequently result in higher exploitation and a higher number of visits. Therefore, the number of visits should be used to make the decision on the best move.

Chapter 4

Technical Implementation

The game was implemented in Java, a platform-independent programming language. It offers many extensions to develop such a complex game. The aspect that Java is a object oriented programming language provides many advantages to model useful classes, which can demonstrate the structure of such a game. Java is also fast enough to calculate all dependecies and to return the user a response on their interaction with the game in a suitable time. But therefore isn't a high-end computer needed. In this context no frameworks were used. The graphical user interface was realized with the assistance of the GUI-Library SWING for Java, which is based on the Abstract-Window-Toolkit (AWT).

4.1 implemented Features

The following section describes the features, which were implemented.
After the game was started, the main menu will be opened. The user can start a new game, make settings and exit the game in this menu. Within the settings menu the user is able to define how many player are part of the game (from one to three) and which sort of player each defined player is (human player, random AI or UCT AI). When a player is a UCT AI an additional option is available, namely, which level of difficulty the UCT AI has.

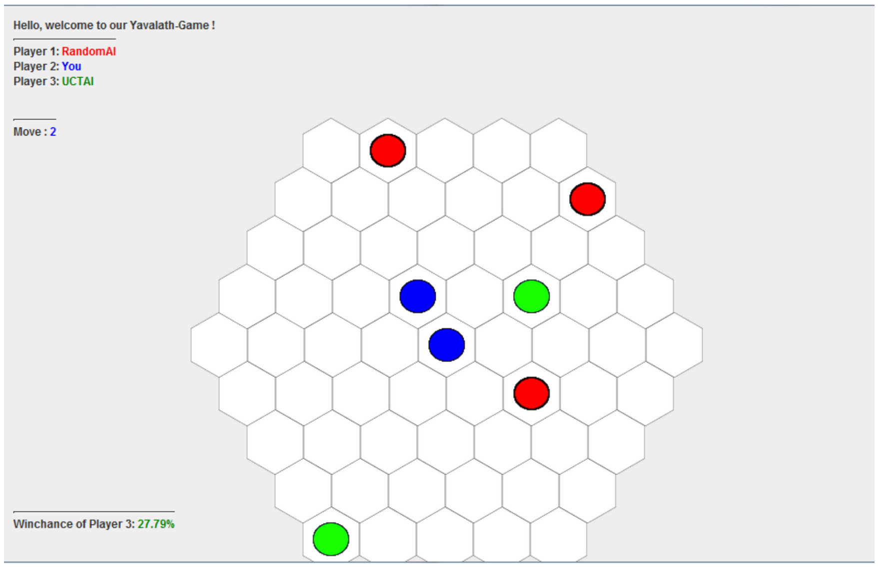

Figure 4.1: Playground of Yavalath implementation

Figure 4.1 shows an example of a typical playground of the Yavalath implementation. The upper left corner shows some general game information. Which player has which color and what kind of type the different players are. Beneath this information the field 'Move' shows, which players move it is, with the specific color. The field in the lower left corner shows the calculated winchance of the UTC AI, presented in percent. If there are multiple UCT AI in the game, the certain UCT AI can be indicated by the reference to the color.
The main part of the window is used for the real playground. When it is a human players turn, the player is able to click on the court, where he wants to set his token.

4.2 Implementation of the UCT-AI

The implementation of the UCT-AI is primarily based on the theoretical foundation of the Monte Carlo Tree Search and the UCB1 policy explained in the previous chapter. This section deals with implementational details such as how the algorithm was tweaked and how the implementation differs from the theory. Sensei's Library features pseudo code for the UCT algorithm which was used as a basis.[1]

The selection phase of the MCTS uses the UCB1 policy as described before. The main adjustment done in this part is the choice of the constant C in the UCB1 policy from equation 3.1. In Sensei's Library, it is suggested to use a value of $\frac{1}{sqrt5}$.[1] As this is a smaller number than $sqrt2$ suggested by Auer, it results in a more exploitational tree search. Therefore, good moves are investigated more than exploring the total width of the tree. It turns out that this works very well with Yavalath.

The expansion phase is slightly modified. Firstly, when expanding the tree by a new layer, all children nodes of the currently expanded node are added to the tree. From the new nodes, a random node is selected. If the path to the newly added layer is chosen again, each of the nodes is visited at least once before a definite selection by the UCB1 policy is made. The Monte Carlo Tree Search suggests to expand the tree as soon as a leaf node is reached. This was modified by the fact that each node needs to be visited at least 10 times before it is expanded with its children. This means that the simulation phase would start right after the selection phase if a leaf node has less than 10 visits. The reason behind this is to reduce to size of the tree stored in memory and therefore make the algorithm less prone to run out of memory. The number 10 was again suggested by the Sensei's Library and works very well in this regard, too.

The simulation phase is done with the help of the Random-AI described earlier. Depending on the number of players, the same number of Random-AI's finish the game. This phase is the main reason behind some AI logic implemented in the Random-AI. If the Random-AI would choose its moves completely randomized, it would also reduce the effectiveness and strength of the UCT-AI. The reason behind this is that games at the later stage tend to have lots of fields on the board where a player can lose by placing a third token in a row. The problem here is that a human player would most likely never do such a move if it is not forced. Therefore, many simulated games would result in not representative outcomes, even though it is way more interesting how the game turns out if these moves are avoided. The same holds for following suit of a forced move or taking a winning move which is also done by the Random-AI. Having this added logical improvements in the Random-AI, the simulations done in the UCT algorithm become much more valuable and representative so that the overall AI increases in strength.

For the backpropagation phase, there is nothing to adjust or modify. It is only important to keep in mind that for each player the outcome of the simulation is different (expect for a draw). When having two players, the result of the simulation has to be alternated on each layer / node when doing the backpropagation. Accordingly, the result has to be matched with the player of the current node when dealing with three players.

Finally, there are two different ways to execute the UCT algorithm. First, one can set a fixed time and let the algorithm do as many iterations of the 4 steps as possible. Second, one can set a fixed number of iterations and have the algorithm run a somewhat undefined time depending on the power of the CPU on which it runs. It was decided to implement the second option, because this enables to define different difficulties (i.e. strengths) of the UCT-AI. The more iterations there are, the more exploitation as well as exploration the UCT-AI could do and thus resulting in a better choice of move. Less iterations mean a weaker AI whereas more iterations make a string AI. Option one with fixed time does not guarantee the same nummber of iterations on different CPUs and therefore, the AI would not perform consistently on every PC. In addition, there should not be a huge difference in execution time, provided that the PC's CPU is not too outdated.

Having experimented and played against the UCT-AI the following number of iteration were chosen for the corresponding difficulties:

- *Easy:* 10000 simulations
- *Medium:* 20000 simulations
- *Hard:* 50000 simulations