Computer science

Variants of Monte Carlo Tree Search

in the game ColorShapeLinks

Koen Oppenhuis

Supervisor:
Mike Preuss

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/08/2022

Abstract

For the game ColorShapeLinks, we will examine different variations of Monte Carlo Tree Search algorithms. The goal is to optimize a Monte Carlo Tree Search (MCTS) algorithm, with the requirements set by a competition for this game; the optimized algorithm will then take part in this competition. First, the main MCTS algorithm is fine tuned, after which some variations are tested. Two of them outperform the main variation slightly and are put together to try and create an even better variation. After some parameter tuning, we will show that this variation is significantly better than the main variation of MCTS.

Contents

1 Introduction **1**
- 1.1 Thesis Overview . . . 1

2 ColorShapeLinks **2**
- 2.1 Rules . . . 2
- 2.2 Example . . . 2

3 Related Work **3**
- 3.1 Connect Four . . . 3
- 3.2 Chess . . . 4

4 Monte Carlo Tree Search **5**
- 4.1 Monte Carlo Tree Search explained . . . 5
- 4.2 Exploration versus exploitation . . . 6
- 4.3 Upper Confidence Bounds for Trees . . . 6

5 Variations **8**
- 5.1 Baseline: UCT . . . 8
- 5.2 UCT Decisive . . . 8
- 5.3 UCT with sufficiency threshold . . . 9
- 5.4 UCT with Move Reduction . . . 9

6 Implementation **11**

7 Experiments **12**
- 7.1 C-value tests . . . 12
  - 7.1.1 Wide tests . . . 12
  - 7.1.2 In depth tests . . . 12
- 7.2 UCT decisive . . . 13
- 7.3 UCT with sufficiency threshold . . . 14
- 7.4 UCT with move reduction . . . 15
- 7.5 Combining sufficiency threshold and move reduction . . . 16
  - 7.5.1 Variable α-value . . . 17
  - 7.5.2 Variable move reduction parameter . . . 18

8 Conclusion **19**
- 8.1 Error calculation . . . 19
- 8.2 Player 1 versus player 2 . . . 19

9 Discussion **20**
- 9.1 Move time . . . 20
- 9.2 Parameter tuning . . . 20
- 9.3 Player 1 versus player 2 . . . 20

References **21**

1 Introduction

Every year, the IEEE (Institute of Electrical and Electronics Engineers) organizes a conference on games. At this conference, competitions are hosted for AI agents to compete against each other. One of these competitions was on the game ColorShapeLinks. In an article [Fac21], written by the organizer of the competition, Nuno Fachada, the following is claimed about this game:
"Machine learning techniques [...] together with a tree search approach such as Monte Carlo Tree Search (MCTS) [...] will certainly be able to produce hard-to-beat agents. However, the limited and well-defined ruleset leaves the door open for knowledge-based or even analytical solutions."
I find the idea of an agent that can play the game via a Monte Carlo structure (random play-outs) very interesting. This is one of the reasons why I wanted to research Monte Carlo Tree Search. To do this in a competitive setting is an extra stimulant for me to try and create an agent, which plays the game as good as possible. At the end of the thesis, I hope I can say that this algorithm outperforms other algorithms, and maybe also human players.
Another reason why I was interested in this competition, was because of the game, ColorShapeLinks. ColorShapeLinks is based on the game Simplexity[Gam]. It seems quite simple and looks like Connect 4, but because of one influential difference in rules, the game becomes much more complex. More on on this can be read in Section 2 and Section 3. Although an algorithm like MCTS is often used to try and play games, with enough understanding of the algorithm, they can also be used in real world problems. Therefore there can be valuable lessons learned in trying to create an optimal agent for a certain game.
The creating of an optimal agent is also the goal of this thesis: What optimizations can be made to a MCTS algorithm to try and play the game ColorShapeLinks as good as possible.

1.1 Thesis Overview

This thesis begins with an explanation of the rules of the game ColorShapeLinks in Section 2. In Section 2.2 you will also see an example of an interesting game state. In Section 3, I will look into the use of MCTS in similar games as ColorShapeLinks. I will explain a bit more about ColorShapeLinks strategy as well. Section 4 gives an explanation about the algorithm, Monte Carlo Tree Search. In Section 6, we explain a bit about the soft and hard ware we used. In Section 5 we explain something about the different variations of this algorithm that were implemented for our experiments. Section 7 contains all experiments done, and their results. In Section 8, there will be a short conclusion about the performance of the optimized algorithm, and finally in Section 9, some interesting characteristics about the research are discussed.

2 ColorShapeLinks

As said in the introduction, ColorShapeLinks is an implementation of the game Simplexity. The rules of ColorShapeLinks and Simplexity are exactly the same, so from now on I will reference to this game with the name ColorShapeLinks.

2.1 Rules

ColorShapeLinks looks a lot like Connect Four. It is a game for two players and each player has its own color, a player has to connect four pieces of a given type to win and the game is played on a grid like board with 7 columns and 6 rows. The placement of the pieces is also similar, a player always has to place a piece as low as possible in a column. The difference with Connect Four is that pieces are not only defined by color, but also by shape; pieces can be round or square. Player 1, who plays with white and starts, wins when he can connect either four round pieces or four white pieces. Player 2, who plays with red, wins when he connects four square pieces or four red pieces. Both player start with 21 pieces of their color, 11 square and 10 round pieces. To win a game you need to connect 4 pieces of your shape or color. It can happen that there there are four pieces in a row of the same color, and four pieces in a row of the same shape. In that case shape is more important than color. Just as in connect four, it is also possible that there are no pieces left, and the board is full. In that case, the game is drawn. All the rules can be found below in Table[1].

	Player 1	Player 2
Color	White	Red
Pieces	11 white square pieces 10 white round pieces	11 red square pieces 10 red round pieces
Wins with:	4 round pieces in a row 4 white pieces in a row	4 square pieces in a row 4 red pieces in a row

Table 1: The rules of the game ColorShapeLinks

2.2 Example

These rules lead to a game that is way more complex than Connect Four. Take for instance the game in Figure[1]. There are a few things to note in this position. It is player 1's turn to place a piece. Remember: he wins with 4 white or 4 round pieces in a row and he loses with 4 red or 4 square pieces in a row. There is no move for player 1 that wins him the game, but he sees that if it was player 2's turn, player 2 could place a red piece in column 6, and get 4 red pieces in a horizontal line. Of course, player 1 would like to stop this and place a white piece there. Only there is a problem, Player 1 is all out of round pieces, so he can only place a square piece there. Unfortunately, this will not help him, because player 2 then wins with 4 square pieces in a horizontal line.
You can see in this instance that it is important to plan ahead and not be too greedy with placing all of your winning shapes directly at the start.

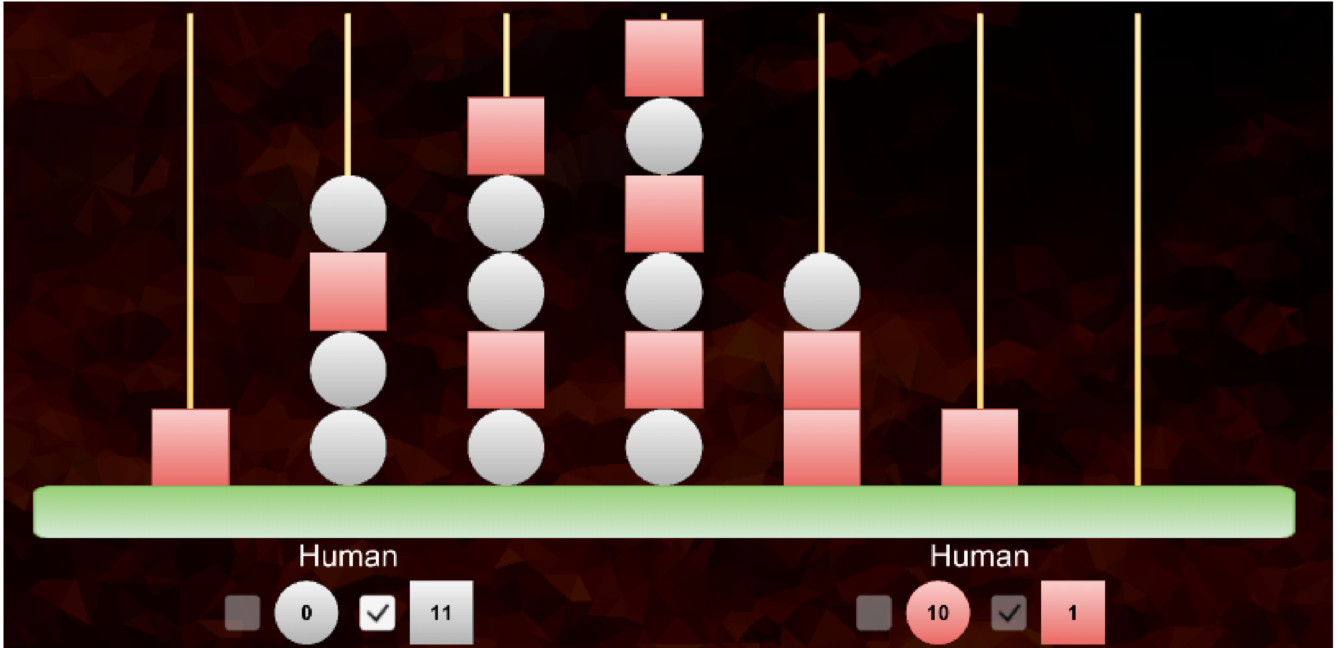

Figure 1: An interesting example game, player 1 (white) is to play

3 Related Work

To try and understand the complexity of ColorShapeLinks, we are going to make a comparison with two other games. See also Table[2].

3.1 Connect Four

Connect Four is similar to ColorShapeLinks, except it has no shapes. This small change makes ColorShapeLinks a lot more complex.
Connect Four was solved in 1988 by V. Allis[All88]. This means that for every game state the best move is known. Allis proofed in his thesis that with perfect play the game is always won by player 1. He used a knowledge-based algorithm, based on nine strategic rules. Each of these rules were proven to be correct.
To get an idea of the complexity of the game, you can estimate the amount of possible positions that can be reached. For Connect Four, every board position can be in one of three states, empty, white or red. So the upper bound for the number of possible positions is 3 to the power of board positions. Just as in ColorShapeLinks, Connect Four has 42 board positions. the upper bound is $3^{42} \approx 1.09 * 10^{20}$. When we compare this to ColorShapelinks, you can see that Connect Four is far less complex. For ColorShapeLinks the upper bound is for game positions is $5^{42} \approx 2.27 * 10^{29}$. $\frac{5^{42}}{3^{42}} \approx 2 * 10^{9}$. This means that the upper bound of the amount of possible positions for the game ColorShapeLinks is around 2 billion times bigger than this upper bound is for Connect Four.
In 1995 John Tromp used brute force techniques to solve Connect Four [Tro]. Because of the big difference in possible positions this is not possible for ColorShapeLinks.
Another way to look at the complexity of a game, is to look at the branching factor of a game. The branching factor is the amount of moves that each player has each turn. For ColorShapeLinks this

initially is $2 \times 7 = 14$(2 shapes, 7 columns). For Connect Four this initially is only 7, the players do not have to think about shapes. This branching factor will eventually get lower for both these games, because columns can get full with pieces, and than you can't place a piece in that column anymore.

3.2 Chess

To compare ColorShapeLinks to chess, we are going to use the same complexity measures. First, lets look at the number of possible positions. In 1950 C. Shannon wrote the following on this subject[Sha50]: *"The number of possible positions, of the general order of $\frac{64!}{32!(8!)^2(2!)^6}$, or roughly 10^{43}"*. Just as in ColorShapeLinks and Connect Four this is an upper bound of the amount of positions. This calculation also includes a few illegal position. For instance positions with pawns on the first rank, or positions with both kings in check.
After calculating the equation of Shannon, the upper bound for the number of positions for chess is $4.63 * 10^{42}$. When we divide this number with the upper bound of positions of ColorShapeLinks you get the following value: $\frac{4.63*10^{42}}{2.27*10^{29}} \approx 2 * 10^{13}$. The upper bound of chess positions is around 2 billion time 10 thousand times as big as the upper bound of ColorShapeLinks positions. This shows that compared to chess ColorShapeLinks is far less complex.
The branching factor of chess is more difficult to calculate than in ColorShapeLinks. This is because in each position there is a different amount of moves. In an article by J. Burmeister and J. Wiles[BW95], they estimate the branching factor for chess to be 35. In contrary to ColorShapeLinks the branching factor can grow or shrink every move and does not have to get smaller.

	Connect Four	ColorShapeLinks	Chess
Amount of possible positions	$\approx 1.09 * 10^{20}$	$\approx 2.27 * 10^{29}$	$\approx 4.63 * 10^{42}$
Branching Factor	≈ 7	≈ 14	≈ 35

Table 2: The complexity of the game ColorShapeLinks compared to Connect Four and Chess

4 Monte Carlo Tree Search

As explained in the previous chapter, it is not possible to use a brute force algorithm on this game. However, the branching factor is not as big as for instance in chess. Therefore this game is very interesting to write an AI-agent for. The AI-agent that was created for this game was using Monte Carlo Tree Search (MCTS).

4.1 Monte Carlo Tree Search explained

Monte Carlo Tree Search is an algorithm that uses random simulations to approximate the value of a certain move. The algorithm progressively builds a partial game tree, based on results of previous simulations. This tree is used to estimate the value of certain moves in further simulations. The bigger the tree, the better the estimate is.
Every node in this partial game tree represents a game state. Every link to child nodes are actions that get you to that subsequent state represented by the child node. If in a certain node the game is finished, then that node is in a terminal state. If a node has unvisited children nodes, than that node is expandable.

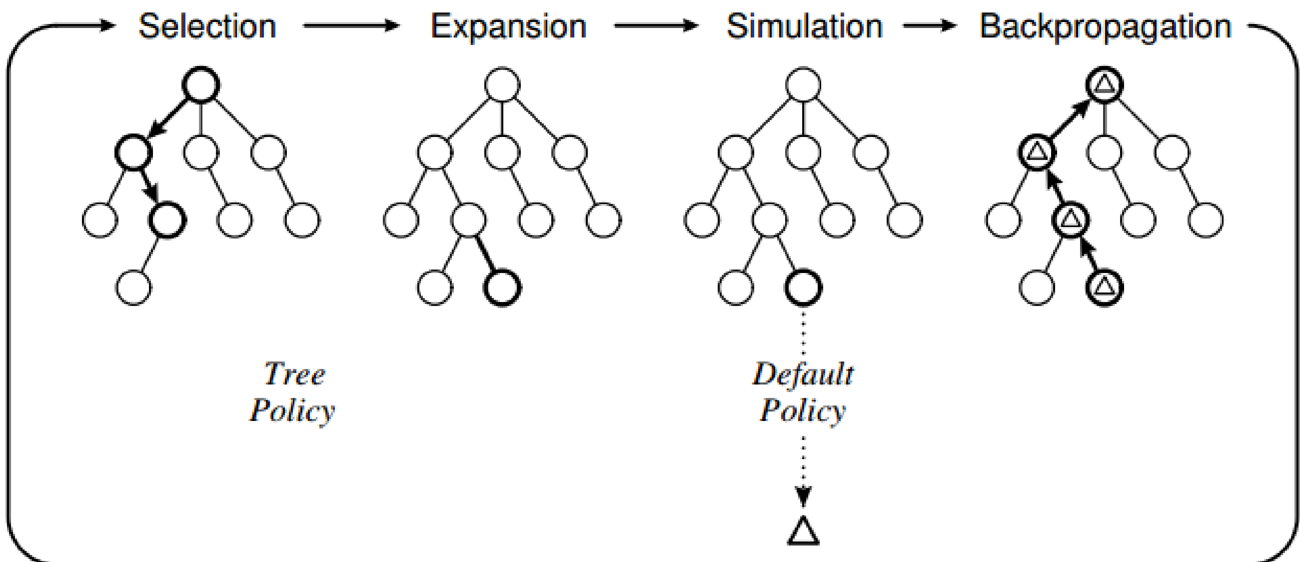

Figure 2: The 4 different stages of a MCTS algorithm

Every iteration of the algorithm makes the tree a little bit bigger. One such iteration of the algorithm can be divided into 4 different actions:

1. Selection
   In the selection phase a node of the tree is selected. This node can not be terminal and has to be expandable. This selection is made recursively from the root, based on the selection policy.

2. Expansion
   From this selected node, a child node is added, based on the available actions from the selected node.

3. Simulation
   From this expanded node, the game is often finished with random moves. For games that are too big for simulation until the end, you can also decide to use different simulation policy's. The important part is, that at the end of the simulation phase a score is given to the selected action.

4. Backpropagation
   The outcome of this simulation (win/loss/draw) is now backed-up through the tree, to update the statistics of all of the nodes that where needed to get to the expanded node. This way the win percentage of each of these nodes is getting higher with a win, and getting lower with a loss.

After a certain amount of iterations are over, a game tree is created with the wins and losses per node. Based on these values the best move is chosen. In order to choose the best move, the algorithm looks at the child nodes of the root of the tree, because that are the moves that can currently be played in the game. Based on the work of Chaslot et al [CWB], there are 4 ways the best move can be selected:

1. The child with the highest win percentage.

2. The child with the highest visit count.

3. The child with the highest win percentage and the highest visit count. If this child doesn't exist, than play more iterations until this child does exist.

4. The child that maximizes a lower confidence bound.

For the MCTS algorithms in this paper, the second option was always used.

4.2 Exploration versus exploitation

Exploration and exploitation are important concepts in MCTS algorithms. It seems best to look at the moves with the highest win percentage that are found by the algorithm, and build further on them. This is called exploitation. However, it is possible that the random simulations for certain moves are not played optimal, and that for instance a move with a low win percentage, actually is quite good. Therefore it is not always best to look at the move with the highest win percentage. The algorithm has to try moves which have a bit lower win percentage too, and see if this move is really bad, or that the simulations where just unlucky. This is called exploration.
This is all controlled by the selection policy that is implemented in the algorithm.

4.3 Upper Confidence Bounds for Trees

The most popular MCTS algorithm[BPW+12] is Upper Confidence Bounds for Trees, UCT. Kocsis and Szepesvari[KS06] first proposed this algorithm. As selection policy they used the UCB1 formula as proposed by Auer et al[AFK02]. They used this formula to optimize a multiarmed bandit problem.

This was also an exploration versus exploitation problem. In the selection phase each child node is chosen by maximizing this following formula.

$$value = winpercentage + C\sqrt{\frac{ln(N)}{n}} \tag{1}$$

Whereby *winpercentage* is the win percentage of the child node, n is the number of times the child node is visited and N stands for the number of times the parent node was visited. C is a constant chosen, bigger than 0. This constant accounts for the exploration in the selection phase. The higher this constant is, the more you explore other moves. If you put C at 0, the second part of the formula becomes 0 and then you are just looking at the highest win percentage in every node. To optimize UCT for a certain game you have to adjust this value. Some games will need more exploration than others. There is no unique value that is best for every game.

5 Variations

In this thesis we want to optimize an MCTS algorithm. To achieve this, we experimented with different variations of MCTS. In this chapter we will introduce these variations and explain how they work.

5.1 Baseline: UCT

The first implementation of MCTS is a normal version of the UCT implementation. For the selection policy formula[1] was used. After the expansion to a new node, the game gets simulated randomly until the game is finished. The result of the game, is back propagated through the tree. A value of 1 if the game is won, a value of 0 if the game is lost, and a value of 0.5 if the game ended in a draw. After a certain amount of time the algorithm is stopped, and the most visited child node is chosen as move to play.

5.2 UCT Decisive

UCT decisive uses the same selection policy as normal UCT. The difference is in the simulation policy of the algorithm. Instead of randomly playing out the game, the algorithm checks every

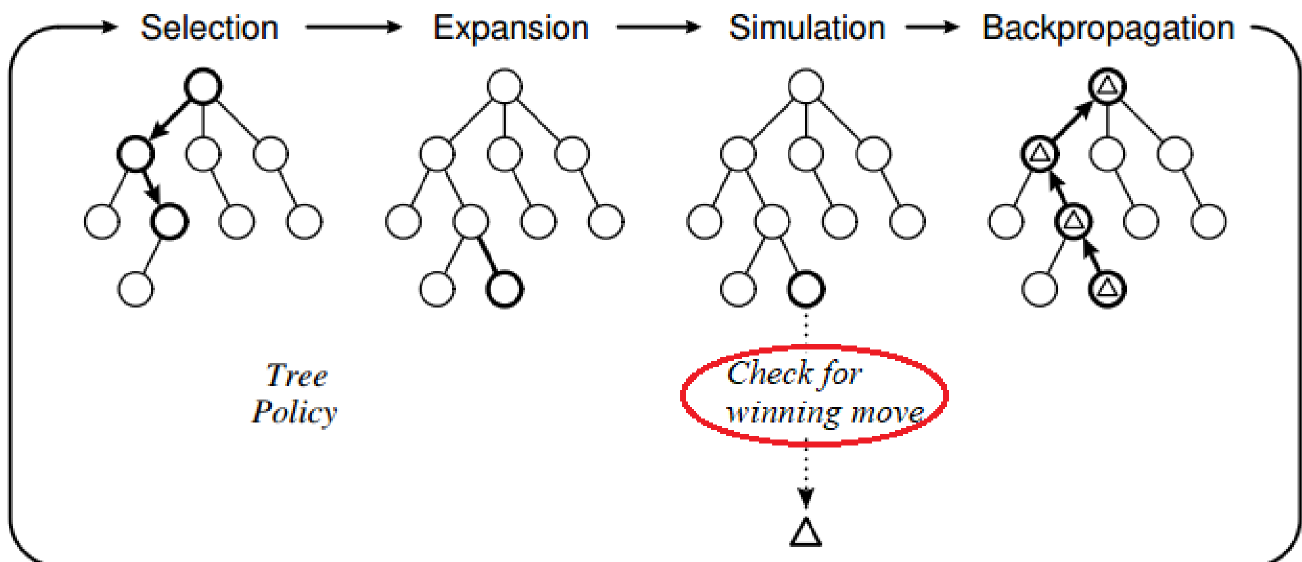

Figure 3: UCT Decisive. In the simulation phase, their is a check for winning moves.

turn, if there is a winning move. If this move exists, it plays this move. This has as advantage that there is no time wasted on non-decisive moves. The downside is that on turns that there is no decisive move, the algorithm takes some time to check if there is a decisive move. We chose to implement this variation because we thought we could save a lot of time in the simulation phase by not skipping winning moves.

5.3 UCT with sufficiency threshold

In this variant the selection policy is partly the same as in UCT. The big difference is that this variant switches from selection policy if the win percentage of a child node is higher than a certain value. To do this we can replace C in formula[1] with Ĉ, whereby Ĉ equals to:

$$\hat{C} = \begin{cases} C & \text{when all child win percentages} \leq \alpha \\ 0 & \text{if any child win percentage} > \alpha \end{cases} \tag{2}$$

In this equation α is a variable that determines when a node has a sufficient enough win percentage.

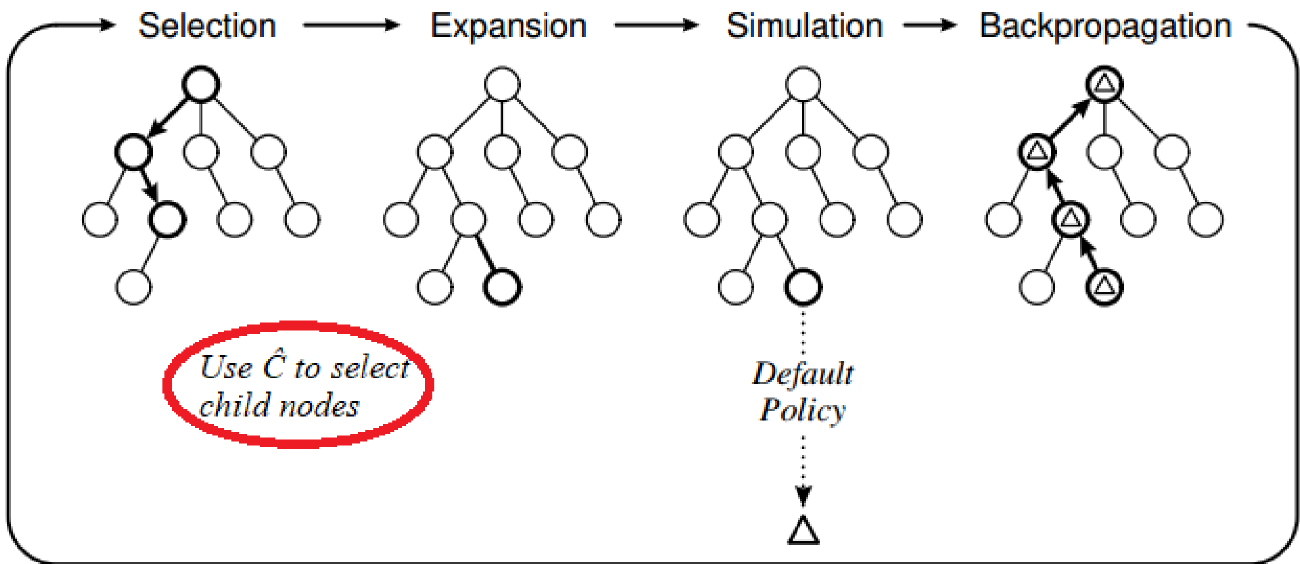

Figure 4: UCT with sufficiency threshold. In the selection phase, Ĉ is used to select the best move.

If there are multiple good moves in a certain position, it often doesn't matter what move you play. Instead of exploring all of these moves, it is better to exploit one of them. It could be possible that a move seems good but after some more iterations drops off below the threshold. At that moment the algorithm goes back to the original UCT method. In the game ColorShapeLinks, every move with your own winning shape seems good in the beginning, but if you place your winning shape too often, then eventually you do not have these shapes anymore. This could lead to a situation seen in Figure[1]. With this implementation we hope the algorithm avoids this situations easier.

5.4 UCT with Move Reduction

This variant is targeted at the expansion phase. Instead of expanding every possible move, you can also expand a selection of moves. This decreases the amount of possibilities for each move, and therefore you can do more iterations in less time. The problem with this implementation is the fact that you have to decide what moves you don't want to explore. For this game I have chosen to decrease the amount of starting possibilities.

At the start, every player has 14 possible moves. There are 7 columns, and 2 different shapes to

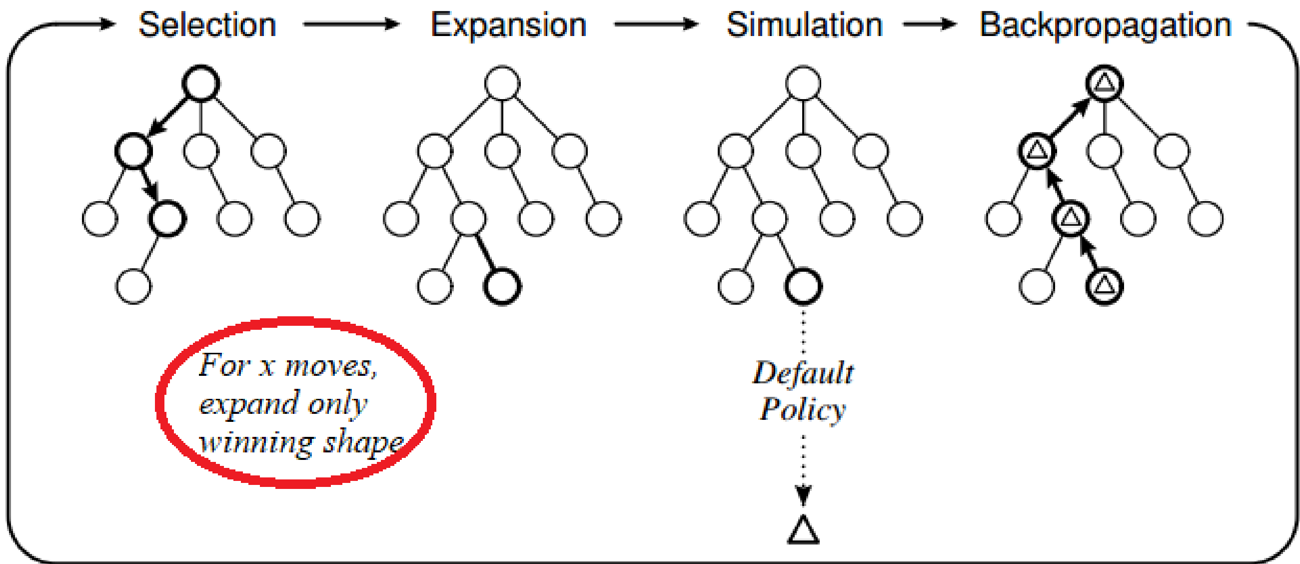

Figure 5: UCT with move reduction. In the expansion phase, for the first x moves, do not expand the moves with the opponents winning shape.

choose from. However, one of the shapes is not very good to start with. As player 1, you wouldn't want to start with a square, because this gives your opponent immediately some initiative, when he places a square piece next to yours. Therefore, for this implementation, the algorithm can only place the shape with which it wins, for the first x moves. Here x is a variable between 1 and 10. The maximum of 10 is chosen because that is the maximum number of pieces player 1 has of its own shape. For simplicity, we set the same limit for player 2.

6 Implementation

These variations were all coded in `C#`. The competition in which this algorithm took part had an own competition framework coded in `C#`. In this framework we could write our own AI-agents and let them play games against each other. For more information on this framework see the following site[1].
All experiments where run on the ALICE cluster of Leiden University. This is to ensure that all played games where under the same circumstances. For these experiments, the normal CPU nodes of the ALICE cluster were used. The normal CPU nodes use two *Xeon Gold 6126 2.6GHz 12 core* processors and have 384 GB of RAM. For more information about the hardware used in the experiments see the following site[2].

[1] https://videojogoslusofona.github.io/color-shape-links-ai-competition/
[2] https://wiki.alice.universiteitleiden.nl/index.php?title=Hardware_description

7 Experiments

Almost all experiments are done with a move timer of 0.2 seconds. This timer was chosen because this was a requirement for the competition in which this algorithm took part. If a different move time is used, then this is explicitly stated in the results.

7.1 C-value tests

7.1.1 Wide tests

In the normal UCT implementation, the only unknown variable is C. For every game a different C-value is optimal, so the first experiments were to determine the most optimal C-value for ColorShapeLinks. This was done by creating a tournament between 8 different UCT-agents with a C-value from 0.1 to 0.8. These values were chosen because of some observations I made while testing the algorithm. With C-values higher than 0.8, because of the short move time, the algorithm would visit every node at about the same frequency. With a longer move time, there is more time for the algorithm to explore, and higher C-values should be researched. Every agent played 30 games against the other agents, 15 with white and 15 with red. Because MCTS algorithms are not deterministic, but based on random playouts, multiple games have to be played to get a better idea which C-value is best. In total every agent played 210 games. This gives a good idea which C-value performs best on average. The win percentage is calculated based on the amount of points each C-value got in this experiment. A win was worth 1 point, a draw was worth 0.5 points and a loss was worth 0 point. The total amount of points was divided by the total amount of games played. The win percentage of each different C-value can be seen in Figure[6].

As you can see, a C-value of 0.3 did best in this tournament. It scored 125.5 point, which is equal to a win percentage of 59.8%. Second best was a C-value of 0.4, with 119 points, and third best was a C-value of 0.2 with 113 points. They had, respectively, a win percentage of 56.7% and a win percentage of 53.8%. As the C-value gets higher, towards 0.8, the win percentage gets less. As discussed, this is because for this C-values the algorithm explores too many options and doesn't exploit certain moves.

7.1.2 In depth tests

To try and further optimize the C-value another tournament was created. This tournament has the same rules as the previous one. Only the C-values were changed to values between 0.22 and 0.36. The previous tournament had some weaker performers, and with this experiment, the values are tested against stronger opposition. This ensures that the C-value found is not just good in exploiting weaker opponents, but also against strong opponents. Again there are 8 different C-values. They all play each other 30 times, so in total every C-values plays 210 games. The win percentage is calculated based on the amount of points each C-value got, with a win being worth 1 point, a draw 0.5 points and a loss 0 point. The total amount of points was divided by the total amount of games played.

The win percentage of the different C-values can be seen in Figure[7].

The C-values in this tournament all have a win percentage around 50%. This is logical because the values do not differ a lot from each other. The C-value that did best this tournament was 0.24. This value had a win percentage of 56.2%. This win percentage was reached against stronger opposition

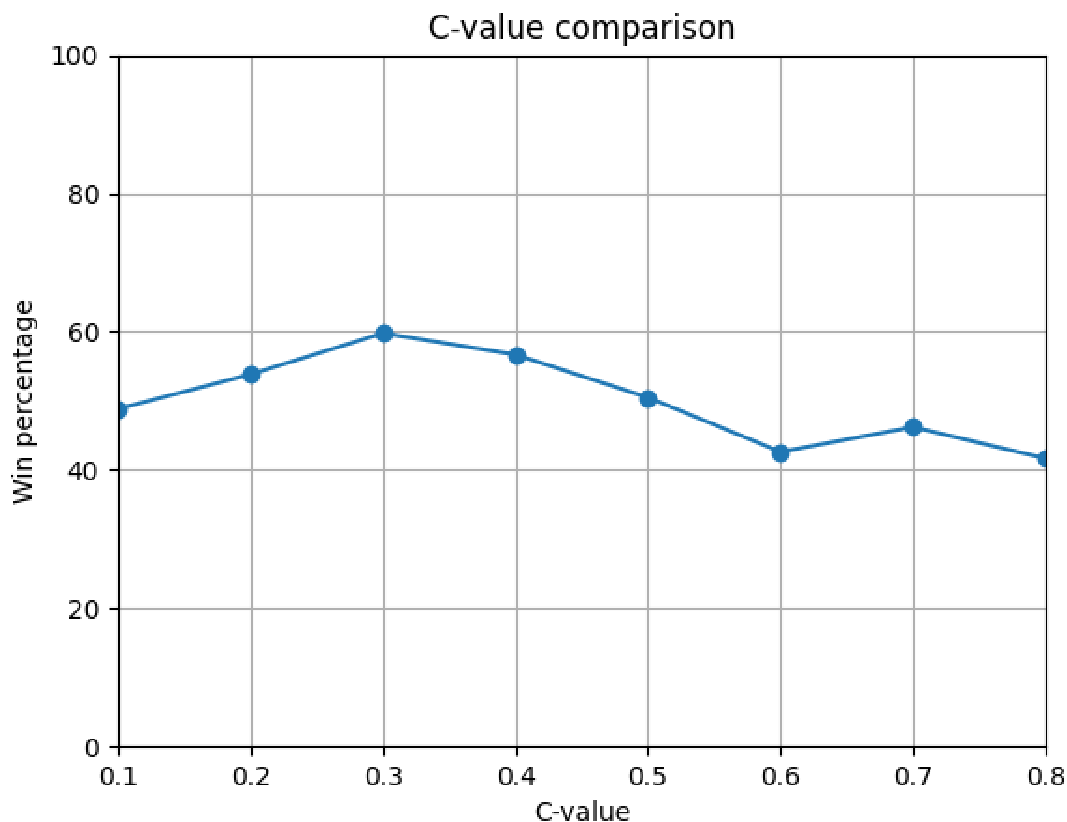

Figure 6: Wide competition: win percentage of a wider range of C-values competing against each other

than the win percentage in the wide tests. That explains why this win percentage is lower than the highest win percentage in Figure[6]. Because this win percentage was reached against a lot stronger opposition than in the previous tournament, this C-value is used in the rest of this paper.

7.2 UCT decisive

To decide if UCT decisive was better than normal UCT, they played 150 games against each other. 50 were in a short time format, with 0.2 seconds per move, 50 were in a medium length format, with 1 second per move and 50 were in a long time format, with 10 seconds per move. These different move times where chosen, because it was not yet decided to do the research only for the 0.2 seconds per move. I included the results on the other time controls, because they helped me in deciding if this implementation of UCT was better than the normal UCT implementation. You can see the results below in Table [3].
In these games, it became obvious that UCT decisive could do way less iterations per move than normal UCT, because before every move, it had to check if there was a winning move. On the short time control, it was the least noticeable in the results. Of the 50 games, UCT decisive won 22 games, and normal UCT won 28 games. For longer time controls, this became more clear. In

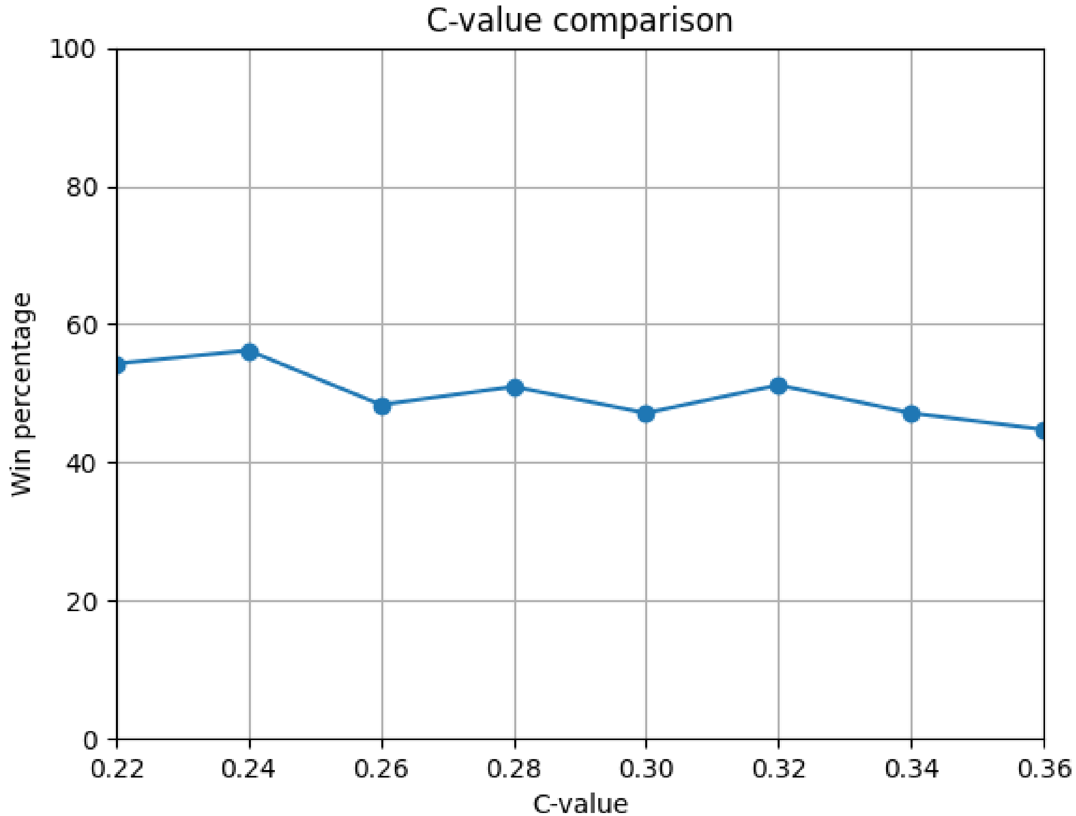

Figure 7: In depth competition: win percentage of a small range of C-values competing against each other

these games the difference in quality was a lot bigger. Of the 50 games with 1 second per move, UCT decisive only won 14 times, and of the 50 games with 10 seconds per move, UCT decisive only won 8. Although the main focus of this research is on the 0.2 second time control, the other time controls helped by making the decision to discard this variation for playing this game.

Time(sec)	0.2	1.0	10
UCT wins	28	36	42
UCT decisive wins	22	14	8

Table 3: UCT decisive versus normal UCT

7.3 UCT with sufficiency threshold

To determine if UCT with a sufficiency threshold is better than normal UCT, 9 UCT-sufficiency agents, with different α-value, played each 30 games against normal UCT. The α-value determines, at which win percentage, the algorithm should stop with exploring different moves, and just focus on the best move and check if it is really good. The α-values in this experiment were between 0.58

and 0.82.
In Figure[8] all the win percentages of each different α-value is plotted.

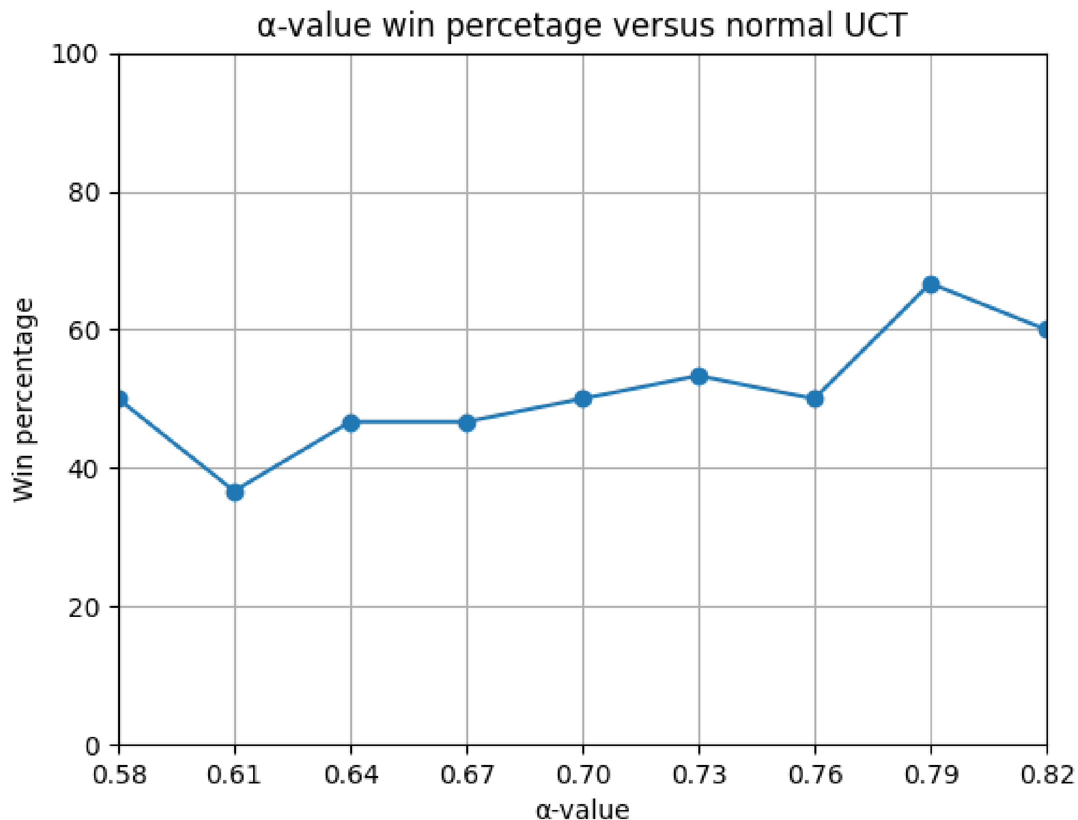

Figure 8: UCT with sufficiency threshold versus UCT: win percentage of different α-values

For the lower *alpha*-values, UCT with sufficiency threshold performs worse than normal UCT. This could be because the algorithm settles for sub-par moves instead of exploring other moves that could be better. In the higher α-values, 0.79 and 0.82, you see an actual improvement over normal UCT. For an α-value of 0.79, the UCT-sufficiency agent won 20 of the 30 games, which equals to a win percentage of 66.7%. This is certainly an improvement over normal UCT.

7.4 UCT with move reduction

For finding if move reduction would improve the UCT algorithm, the same setup was used as for the sufficiency test. For each agent the amount of moves where they only have the choice of 1 shape, is a value between 1 and 9. In figure 9 the win percentage for all the different move reduction parameters is plotted.
If two normal UCT algorithms play against each other, they will also always play their first move with their winning shape, and also often their second move. Therefore I would expect that move reduction for the first 2 moves would always be better than normal UCT, because you eliminate half

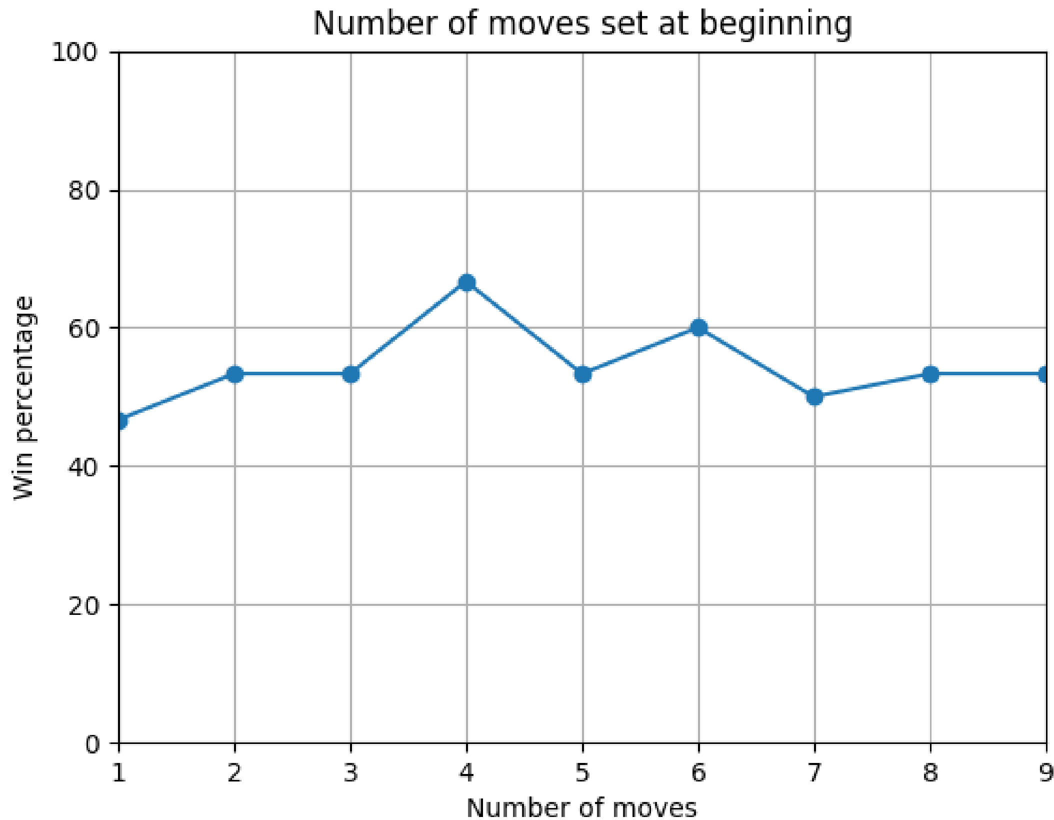

Figure 9: UCT with move reduction versus UCT: Win percentage of different move reduction parameters

of the options on the first two moves. However, in the experiment you can't see a big improvement for parameter 1 and 2. This is possibly because this move reduction only has impact on the first 2 moves of the UCT algorithm. This is just a small part of 1 game.
At higher values, you can see the the move reduction has more impact. Because the algorithm has less choice in the beginning, it can do more iterations, and gather more information that can be used for the entire game. It is a bit unclear if the parameter can be too high. At some point, it isn't always better to place a piece of your winning shape. In this experiment, for high values of this parameter, it performs around equal to normal UCT.
The best value found in this test was a middle value, with a move reduction parameter set to 4. This variation won 20 of his 30 games against normal UCT, this is a win percentage of 66.7%.

7.5 Combining sufficiency threshold and move reduction

If you look at the variants of the UCT algorithm, the sufficiency threshold and the move reduction have a positive effect for certain parameter values. Therefore it seems logical to try and combine these 2 variations. Different combinations of α-values and move reduction numbers where tested. These combinations were chosen based on the outcomes of the previous experiments.

7.5.1 Variable α-value

For the first experiment the UCT-combination agent had a move reduction value of 5. This was not the best value in the movement reduction test, but in Figure[9] you can see that the values of 4 and 6 did really well. Therefore, I think that the movement reduction parameter of 5 is actually quite good as well and got a bit unlucky in the previous experiment. There is no reason that specifically at a move reduction parameter of 5, the win percentage goes down, to then go up again at a move reduction parameter of 6.
The α-values were set between 0.65 and 0.85, because in between these values, the highest win percentages were found, see Figure[8]. Every agent played 40 games versus the normal UCT-agent. In Figure[10] below you can see win percentage of each agent:

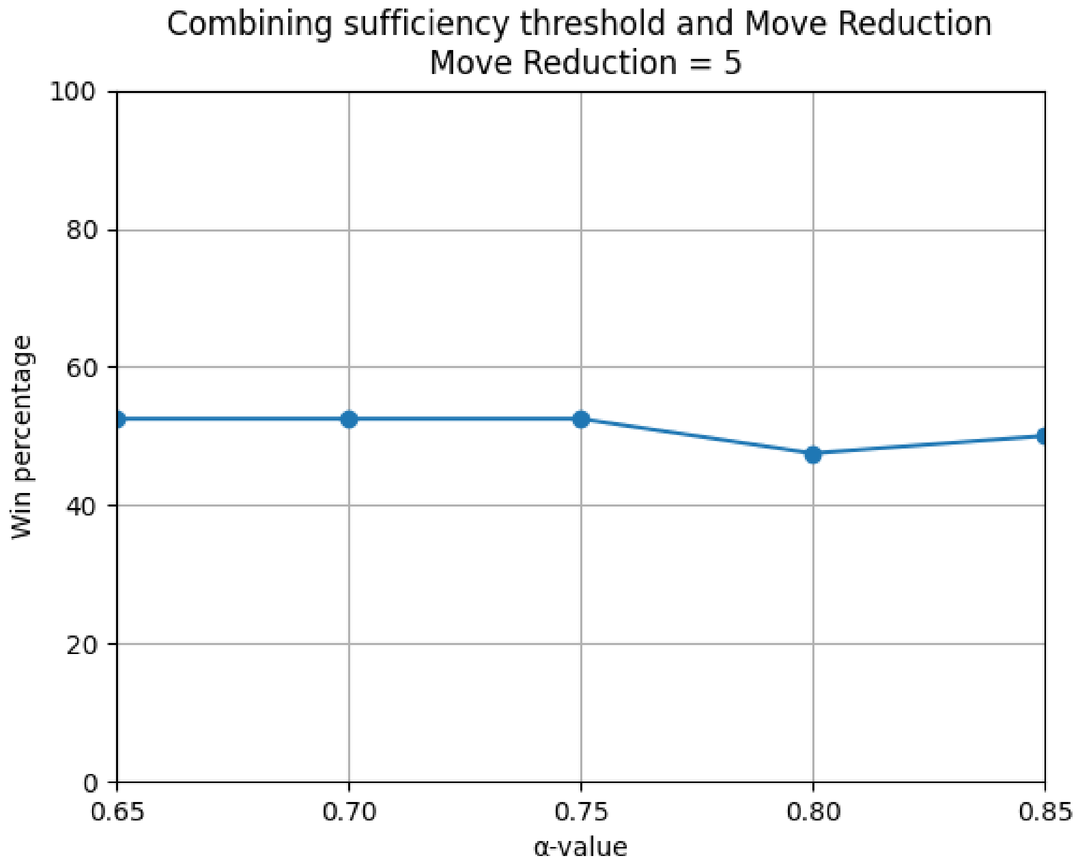

Figure 10: UCT Combined versus UCT: win percentage of different α-values

The results in Figure[10] seemed odd. It appears that the agents don't perform better than normal UCT. The α-values of 0.65, 0.70 and 0.75 all got 21 points of the 40, 0.85 got 20 points and 0.80 got only 19 points. Based on Figure[8] and Figure[9], I would expect a win percentage of atleast around 60%. It could be possible that the sufficiency threshold and the alpha-value don't work together properly, but I do not see a good reason why this should be the case.

7.5.2 Variable move reduction parameter

To check the combination of sufficiency threshold and move reduction again, a similar experiment was done, but now with a variable move reduction parameter. The α-value was set at 0.75. This seemed the best value if you combined the results of Figure[8] and Figure[10]. Based on the best values in Figure[9], the move reduction parameter was set between 3 and 7.
In Figure[11] you can see the win percentage of each agent: In this experiment you see that

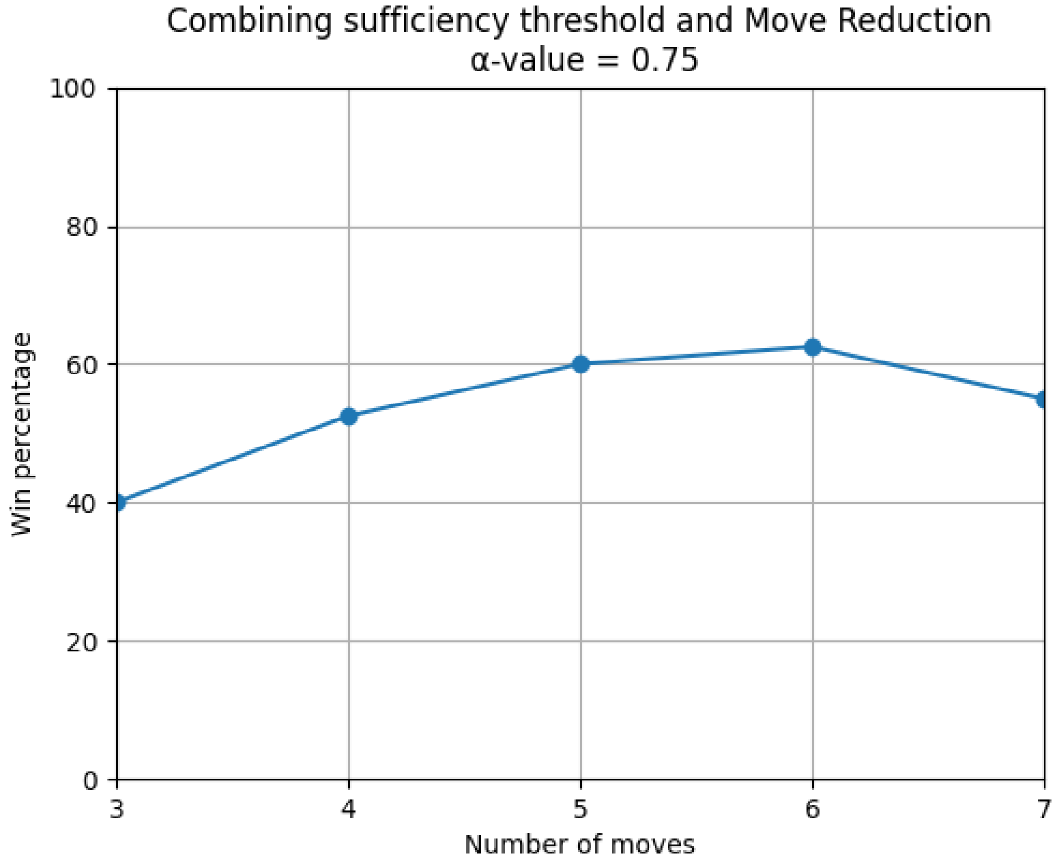

Figure 11: UCT Combined versus UCT: Win percentage of different move reduction parameters

a combination of sufficiency threshold and move reduction can be better than normal UCT. Interestingly, in this experiment the agent with α-value of 0.75 and move reduction parameter of 5 got 24 out of 40 points. This is better than in the previous run of 40 games against normal UCT. This shows that on this time control, an MCTS algorithm still has a great deal of randomness in it. In this experiment a move reduction parameter of 6 scored best, with 25 points in 40 games. This equates to a win percentage of 62.5%.

8 Conclusion

In the end, the combination of these modifications is better than the normal UCT version for the game ColorShapeLinks with a move time of 0.2 seconds. In the final experiment, the UCT agents with an α-value of 0.75 and move reduction parameter of 5 and 6, scored a combined 49 points out of 80 games. If we assume that the UCT agents with modification are as good as normal UCT agents, and that the odds of winning a game are 50%, then you can calculate the chance of winning at least 49 games out of 80.

8.1 Error calculation

First we are going to calculate the chance of winning exacly 49 out of 80 games. You can do this with a binomial equation[3].

$$P(X) = \frac{n!}{(n-X)!X!} * p^X * q^{n-X} \tag{3}$$

Where:

- n: total number of games
- X: total number of wins
- p: chance of winning
- q: chance of losing

The probability of exactly winning 49 games out of 80 is:

$$P(49) = \frac{80!}{(80-49)!49!} * 0.5^{49} * 0.5^{80-49} \approx 0.0118$$

But we want to know the chance to win at least 49 out of 80 games. Therefore we also need to know the chance of winning 50, 51, ...,79, 80.

$$\sum_{X=49}^{80} P(X) = 0.02833$$

The sum of all these values equals 0.02833. This means that the chance to win 49 out of 80 games is only 2.83%. This seems significant enough to say that the modifications made, make the normal UCT algorithm better. Of course we make some assumptions here. For instance that the odds of winning a game between two equal UCT algorithms is 50%. This is actually not entirely true.

8.2 Player 1 versus player 2

As final experiment two normal UCT implementations played 725 games against each other. The results of these games are found in table[4]. Of all games, player 1 only won 244 games, player 2 won 480 of the 725 games and only one single game ended in a draw.

Because in every experiment the UCT agents each played equally often as player 1 and player 2, you could say that eventually the chances to win level out to 50%. For a single game however, the chances of winning as player 2 are 66.2%, and the chances of winning as player 1 are 33.7%.

	player 1	player 2
Wins	244	480

Table 4: Amount of wins between player 1 and player 2, both played by an UCT agent

9 Discussion

9.1 Move time

For such a short move time, it is really difficult to make a big impact on the UCT algorithm. This is because the algorithm is still very random in the beginning iterations. In the first 0.2 seconds, the algorithm is more Monte Carlo like. There is not enough time to converge to an optimal move. Therefore, it could be more beneficial for further research to focus more on longer time controls. Also a point for debate is whether you should use real time to determine how long the algorithm can use for a move. You could also measure the move time by the amount of iterations the algorithm has made. This could be beneficiary to better recreate the experiments performed in this thesis. The disadvantage is of course that one implementation can take a lot more time than another. For instance the UCT decisive algorithm would take a lot of time.

9.2 Parameter tuning

With each different implementation of the UCT algorithm, different parameters could be optimal to use. For this thesis, I chose a C-value of 0.24 for every agent. This can of course be influenced by the modifications made to the algorithm. For instance if a modification is made that helps the algorithm with exploring more options, then it might be better to change the C-value to exploit more.
It would also be interesting to see if the C-value changes if the algorithm could take more time per move. If the algorithm has more time, maybe there is also more time for exploring more, instead of instantly having to choose a good move.

9.3 Player 1 versus player 2

It is interesting to see that with normal UCT agents, player 2 wins 66.2% of games. The parameter used for player 1 could be not optimal if you are player 2. This can not be said for a game like chess, because a position reached in chess can often also be reached from the other side. For this game however, the board will never be similar for player 1 and 2. Player 1 will always see an even amount of pieces on the board, when it is his move, and player 2 will always see an odd amount of pieces. Also the division of pieces is different. Player 1 has 10 pieces of his winning shape, and player 2 has 11 pieces of his winning shape. Therefore it could be possible that the different players need different parameter settings.

References

[AFK02] Peter Auer, Paul Fischer, and Jyrki Kivinen. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, 2002.

[All88] Victor Allis. A knowledge-based approach to connect-four. the game is solved: White wins. In *Master's thesis, Vrije Universiteit*, page 4, 1988.

[BPW+12] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, and et al. A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI*, 2012.

[BW95] J. Burmeister and J. Wiles. The challenge of go as a domain for ai research: a comparison between go and chess. In *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95*, pages 181–186, 1995.

[CWB] G. M. J. B. Chaslot, M. H. M. Win, and B. Bouzy. Progressive strategies for monte-carlo tree search.

[Fac21] Nuno Fachada. Colorshapelinks: A board game ai competition for educators and students. *Computers and Education: Artificial Intelligence*, 2, 2021.

[Gam] Brain Bender Games. Simplexity, https://boardgamegeek.com/boardgame/55810/simplexity (2009).

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[Sha50] Claude E. Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[Tro] John Tromp. John's connect four playground, https://tromp.github.io/c4/c4.html.