MONTE-CARLO TWIXT

Janik Steinhauer

Master Thesis 10-08

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
17 June 2010

Preface

This master thesis is the product of a research project in the program "Artificial Intelligence" at Maastricht University. The work was supervised by Dr. Mark H. M. Winands. During the study, he gave lectures in the courses "Game AI" and "Intelligent Search Techniques". Especially the latter one concentrates on how to build a strong AI for classic board games, most often two-player perfect-information zero-sum games. In the game of TwixT, few serious programs have been written that play at the strength of human amateurs. The probably most successful program so far is based on pattern matching. So it was an interesting question if it is possible to build a strong AI using a search-based approach, i.e., Monte-Carlo Tree Search. Further, it was not obvious how and which pieces of knowledge can be included in the search. So it has been an interesting research project with several surprising results and many challenging problems.

I want to thank all people who helped me with my master project. Of greatest help to me was Dr. Mark Winands, who gave me a lot of feedback on my ideas. He helped me with his experience in with intelligent search techniques whenever it was helpful at difficult decisions. He was also very enduring in correcting my language mistakes, which tend to be repeated again and again. I further thank Kevin Moesker who willingly provided me with the results of his research on TwixT. With his help, I was able to evaluate the result of my own work. I also thank my friends, who patiently listened to me when I tried to explain what exactly I was doing in my master project.

Janik Steinhauer
Aachen, April 2010

Summary

This thesis investigates how to construct a strong AI player for TwixT. We tried to build an AI player for TwixT that plays as strong as possible. TwixT is a perfect-information two-player zero-sum connection game and has a high branching factor, resulting in a high search space. To tackle this problem, we implemented a Monte-Carlo Tree Search (MCTS) player and enhanced it with domain-independent enhancements as well as with knowledge. The result is a player with the strength of a human beginner.

The following domain-independent enhancements contributed to the strength of the program: Progressive History values (a variant of Rapid Action Value Estimation (RAVE)), MCTS-Solver, reusing the MCTS tree in the next search, and a discount rate for the backpropagation of MCTS. The latter two enhancements were newly introduced and have not yet been implemented in other domains. Reusing the MCTS tree is a simple idea that works by saving the search tree until the next search begins. Then, the next search is not started with an empty tree, but with the subtree that corresponds to the new board state. Thereby, a part of the old tree can be reused and gives a better start for the next search. The discount rate in the backpropagation is based on the idea that the results of older simulations are less realistic than the results of newer simulations. Thus, we decided to multiply the counter and the score with a discount rate slightly below one every time that a node is updated.

After implementing most of the domain-independent enhancements, we began including knowledge. As usual in the MCTS framework, the knowledge was included in the selection and in the playout. The most important part of the knowledge is the concept of bridges in TwixT. Preferring moves that allow bridges in the playout by multiplying them with a certain weight was the piece of knowledge that helped the most to increase the playing strength. Some other ideas have also been useful, such as excluding known weak moves.

During the implementation of the different enhancements and features, parameters had to be tuned. As the results show, this is an important step. We decided to optimize the parameters on a 12×12 TwixT board, giving each player 10 seconds to decide for the next move. In this setup, the final program can run 2,100 simulations per second (starting from an empty board), so the final decision is based on 21,000 simulations.

Contents

Chapter 1

Introduction

In this chapter, a brief description of Game AI and an introduction to TwixT are given. The TwixT rules will be described and discussed, especially with regard to Artificial Intelligence. Related research is covered and four research questions are stated and described.

Chapter contents: Introduction – Game AI and TwixT, TwixT Rules and Challenges, Related Research, Problem Statement and Research Questions, Thesis Outline

1.1 Game AI and TwixT

Developing an Artificial Intelligence (AI) for a game has been a popular application for a few decades. It has always been an interesting question whether computer programs can beat human experts in certain games, e.g., in classic board games or in modern computer games. Game AI found its beginning in the 1950s, when Shannon (1950) and Turing (1953) wrote the first articles about chess AI. Since then, AI players have been programmed for many other games, usually with satisfying success. Chess, for instance, has been investigated so intensively that the best chess programs are even better than the best human players in the world (Hsu, 2002). Other games have been considered including board games with rather different rules and structures, for example checkers and Go.

Most of the time it is not possible to make computers "understand" strategies in computer games as humans do. Developers have to utilize something computers are quite good at – doing much computation in a short amount of time. However, the game knowledge in the AI is mostly limited, especially in comparison with human players. Developers have to take care of the computation time of their applications – a search algorithm that makes use of much knowledge, but needs too much time for this, will not produce better results than one that works very simple, but is fast enough to compensate for it by exploring a bigger part of the state-space.

Some board games, such as Go, Havannah or TwixT, have been a real challenge for game developers. In these games, each step provides many possibilities as a next step, resulting in a large search space. TwixT, one of these games will be addressed in this thesis. It was invented by Randolph in 1960. The high number of moves possible in TwixT may be the reason why there are no TwixT programs that are a serious opponent against a human expert. The program considered to be the best is currently the T1J by Schwagereit (2010), which applies a pattern-based approach. In the next section we discuss the reasons for the difficulties regarding a TwixT AI.

1.2 TwixT Rules and Challenges

In this section, the rules of TwixT are described - especially with regard to computer TwixT. Further, some of the difficulties due to these rules are described.

1.2.1 TwixT Rules

There are different rule sets for TwixT. The one described here is the usual rule set for computer TwixT and is used in most programs (Schwagereit, 2010). It differs from the rule set delivered with the original board game, which e.g. allows link removal. In addition, the rule set used here is a lot easier to implement than the original one.

TwixT is a two-player perfect-information zero-sum connection game. It is played on a board with 24×24 holes (except for the corners). The empty board is shown in Figure 1.1. The players perform their moves alternating. The player doing the first move will be referred to as “White”, to the other player as “Black” (although many board game versions and computer programs use other colors. In the pictures the first player is displayed white and the second player black). A move consists of placing a peg into any empty hole on the board (exceptions are described later). Placing a peg into a hole includes placing all possible links (or “bridges”). Two pegs can be linked if they are placed at opposite sides of a 2×3 rectangle, like a knight’s move in chess (e.g., one row up and two columns right). In addition, the new link cannot be placed if it would be crossed by an existing link.

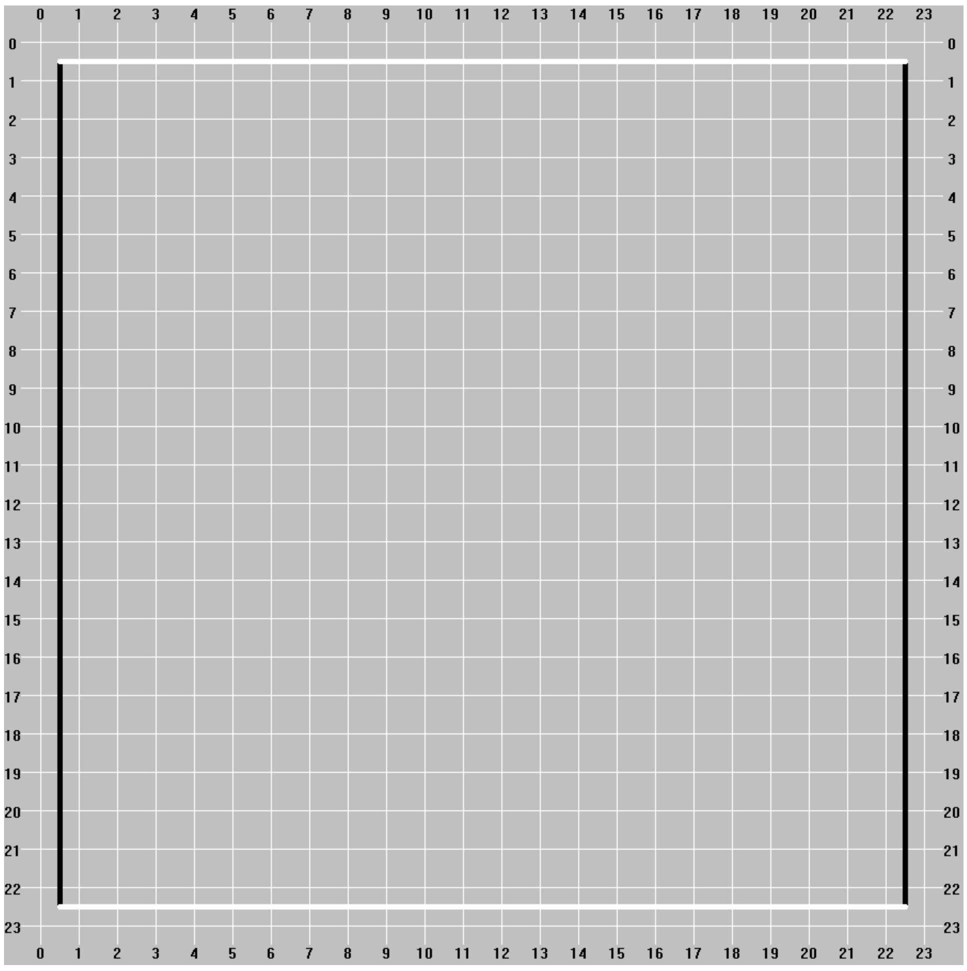

Figure 1.1: Empty TwixT Board

Each player has the aim to connect his two sides: The white player wins if he can connect the left and the right side with an uninterrupted chain of his pegs and links, while the black player has to connect the upper and the lower side with a chain of his pegs and links. The two goals are exclusive, since there cannot be an uninterrupted chain from left to right and at the same time a chain from top to bottom. Figure 1.2 shows a terminal position with the white player (playing from top to bottom) as the winner.

Even a draw is possible, as shown in Figure 1.3. On the one hand, the black player cannot interrupt White anymore, because the end points of White’s two chains are too close to each other. On the other

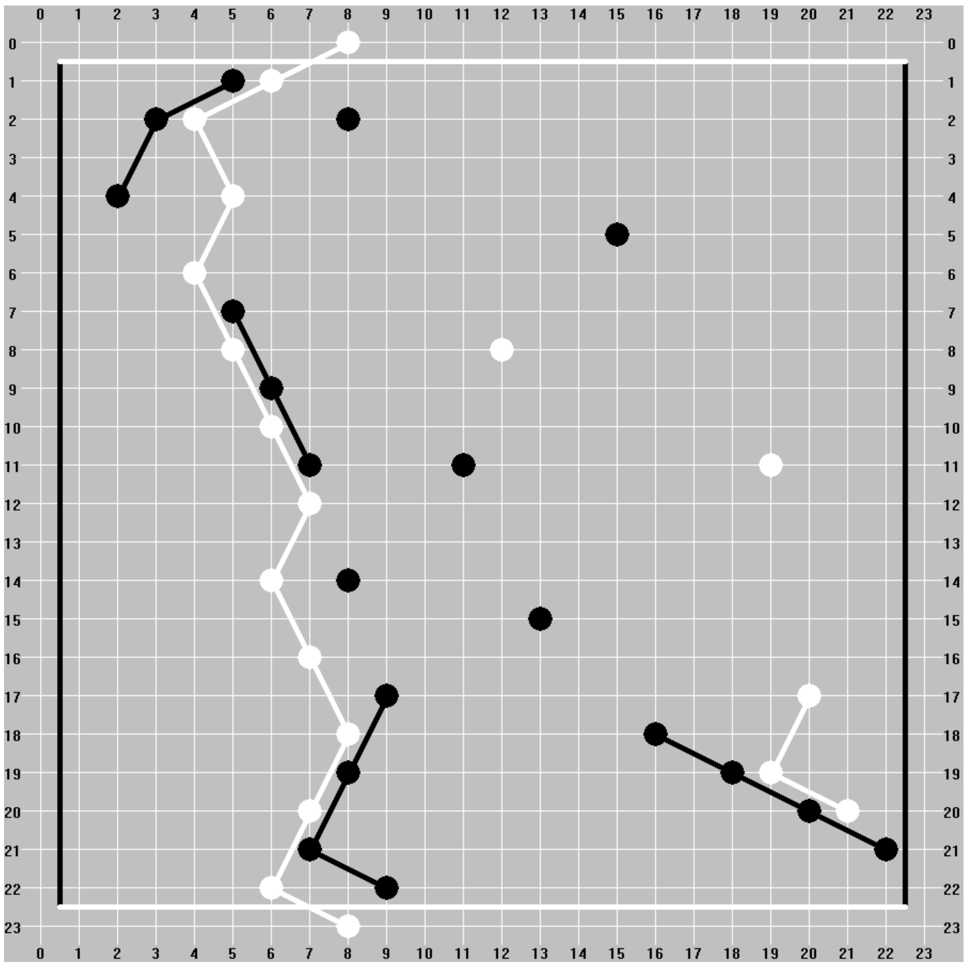

Figure 1.2: Terminal Position: White wins

hand, the white player has no possibility to connect these chains, because Black blocks all moves that would lead to such a connection.

There is one exception for peg placement: No player can place a peg in the "home" of his opponent, so White cannot play the leftmost or rightmost column, while Black cannot play the top or bottom row. This rule seems to be mostly cosmetic, because it leads to the situation that Black usually has already won when he can connect the second column from the left with the second column from the right, because White cannot block anymore. This is almost the same as playing on a board with 22×22 holes. Figure 1.4 shows one of the rare positions where one player reached the line next to his home, but is not able to connect it. Thus, these positions are possible, but will not occur often, neither in a humans' match, nor in a computer match. We remark that if link removal would be allowed the position in Fig 1.4 would be a win for Black.

Usually, the white player has a substantial advantage. He can play somewhere in the center and thereby have quite an advantageous position. To prevent this, Black is allowed to swap sides after the first move, called the pie rule. Thus, White will not make a strong move because Black could steal it. However, White should not make a weak move because then Black will not swap and White keeps its weak move. So the first move will often be an average move.

1.2.2 Difficulties in Computer TwixT

When developing an AI for a board game, we have to investigate the current board position, because all interesting games are much too complex to be solved entirely. So, we have to use heuristic search to find the best move. A well-used and well-working approach for many games is the alpha-beta algorithm (see

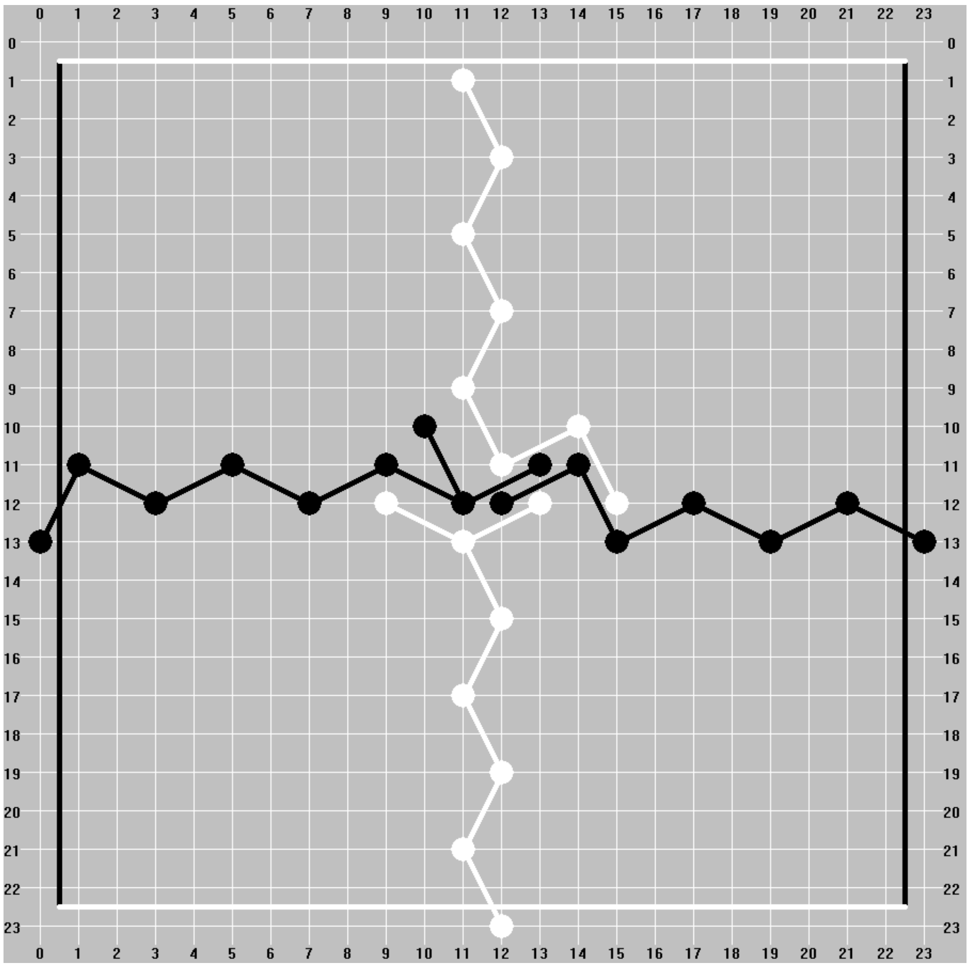

Figure 1.3: Terminal Position Without Winner (Draw)

Knuth and Moore (1975) and Junghanns (1998)), which searches as deep as possible in the search tree of all possible moves to find the best one, using a board evaluation function for the leaf nodes when the maximum search depth is reached. It is possible to develop an alpha-beta player for TwixT as well, but the results of all programs using this algorithm are weak (Moesker, 2009). This is caused by the fact that TwixT produces a large search tree (Moesker, 2009). When playing on a 24 × 24 board, the beginning player has 24 × 22 = 528 possible moves (all holes except for the opponent's home). After each move, this number decreases by only one, so the number of possible moves stays high for many moves. Since a game rarely lasts for more than 100 moves (which is the total amount of pegs delivered with the original game version), the branching factor will rarely decrease to a value below 24 × 22 = 428. This is more than ten times the branching factor of chess and causes problems for the alpha-beta players.

Besides the branching factor, there are other difficulties for the alpha-beta algorithm in TwixT. It is hard to evaluate a specific board position and compare it to others. First, it requires much research to define patterns that would make up a "good" or a "bad" board position. Second, implementing a matching function for these patterns in a program would be work-intensive, because many different pattern instances are possible. Third, after implementing this matching function, it would consume too much time to perform this matching at runtime. These and other factors point to another approach, namely the Monte-Carlo Tree Search (MCTS) (See Chapter 3). MCTS (Kocsis and Szepesvári (2006), Coulom (2007)) is a best-first search which utilizes Monte-Carlo simulations to estimate each move's strength (Chaslot *et al.*, 2008). This method was useful in other games such as Go (Coulom, 2007), General Game Playing (Finnsson and Björnsson, 2008), Phantom Go (Cazenave and Borsboom, 2007), Hex (Cazenave and Saffidine, 2009), Amazons (Lorentz, 2008), and Lines of Action (Winands and Björnsson, 2010).

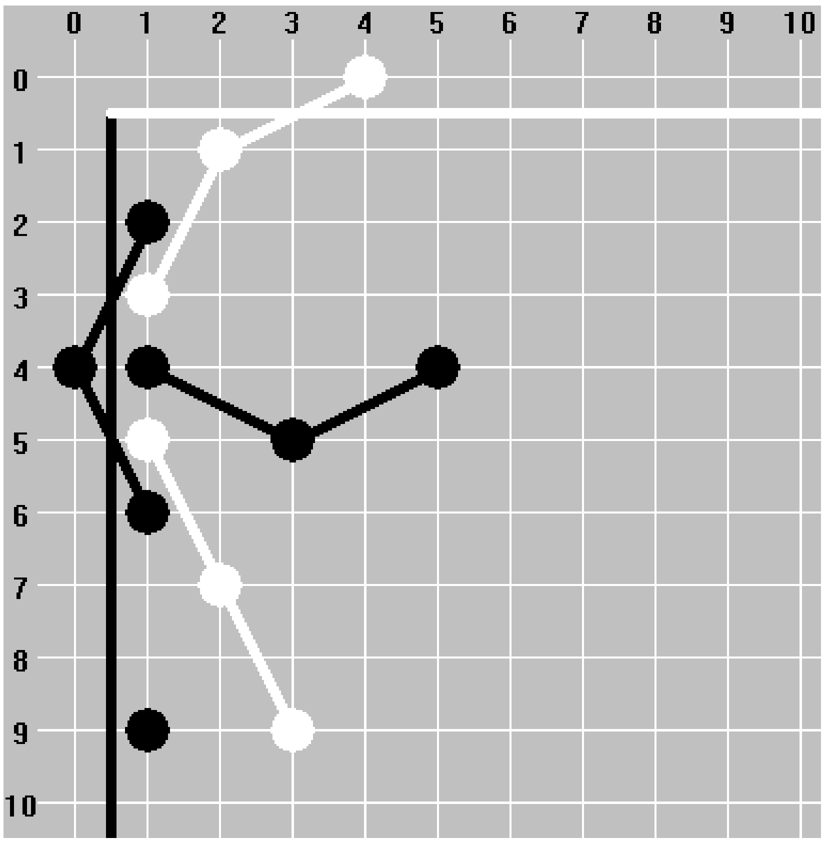

Figure 1.4: Black Cannot Win

1.3 Related Research

This section describes the lessons that have been learned in previous research projects related to the domain of TwixT.

1.3.1 Research on Other Connection Games

Though TwixT itself has not been investigated intensively, other connection games have been. This section is based on Teytaud and Teytaud (2010), a paper about the game Havannah. Research on other connection games also produced some results, which is not covered here (e.g. Hex, see Cazenave and Saffidine (2009)).

From the angle of an AI developer, Havannah has some disadvantages that are similar to those of TwixT. Havannah is connection game where each player can build different types of connections that will make him win the match. There are almost no local patterns known for Havannah, so it is hardly possible to write an appropriate evaluation function for a certain board position. In addition, the search space of Havannah is rather large. Depending on the size of the board that is chosen, the branching factor can be similar to the branching factor in TwixT. Thus, it is hard to find moves that can be pruned because of their low quality. So Havannah is also a game where the alpha-beta algorithm does not perform well. From the research done on the topic, we gain insight about the approaches that will work in TwixT.

One research result of Havannah is the importance of a high number of simulations (in the MCTS framework, see Chapter 3): In Havannah, a player with n simulations played per move will win only about 20 to 25 % of the games played against a player with $2 \times n$ simulations per move. If we assume a similar value for TwixT, we see that we need a high number of simulations as well.

Another lesson that can be learned from the research on Havannah is usefulness of RAVE values (see Section 3.3.1), especially when there is not enough time for a sufficient number of simulations (in ratio to the size of the search space). A player with only this feature can win all games against a player without any features. Unfortunately, the impact of RAVE is less impressive as the number of simulations increases, but it is still significantly improving the playing strength. In addition, the RAVE implementation has to be tuned again as the number of simulation changes, which has to be taken into account in the final tuning of all the parameters.

1.3.2 TwixT Research

Research results on TwixT are still rare. There are some serious attempts to create strong AI players for it, but few of them were successful. The most notable one is the Java program T1J by Johannes Schwagereit (Schwagereit, 2010). It is playing at the strength of a strong amateur player. Unfortunately, we cannot use much of the research results of it because it utilizes a pattern based approach, which requires a high affinity to the game itself, while we want to create a more (general) search based approach.

Another research project on TwixT is Moesker (2009). Its focus was on the complexity of TwixT and some approaches to build AI players - alpha-beta as well as Monte-Carlo. It also included the implementation of two AI players. While the work about the background is quite extensive, the implemented players are not that strong (see Moesker (2009), Chapter 7).

As Moesker (2009) shows, the complexity of TwixT is, in comparison to other board games, quite high. With an average of 60 moves per game (for the 24×24 board), we get an average branching factor of 452 and a game-tree complexity of 10^{159}, which is exceeded by a few games, such as Go. This confirms our assumption that a simulation based approach is likely to perform better at TwixT than a evaluation based approach like the alpha-beta algorithm.

Another part of the research was done on how to implement an alpha-beta player. The alpha-beta player used an evaluation function to estimate the "value" of the current board position, indicating which player is more likely to win. Several different features were suggested. These features could be taken into account when we implement the heuristic values for the simulation (see Chapter 4). But, as we will see in that chapter, the features we use have to be calculated much faster than the features we found in the evaluation function of Moesker (2009).

In contrast to previous work on TwixT, we try to use easy-to-calculate knowledge-based features instead of complex features that require much theoretical work. These light features may produce better results than the complex features.

1.4 Problem Statement and Research Questions

To advance the state-of-the-art in the domain of TwixT, we will work on the following problem statement:

- "How can we implement a Monte-Carlo Tree Search TwixT player that defeats strong human amateur players?"

This leads to the following four research questions:

1. *Which data structure can we use for TwixT in order to achieve the best performance with MCTS?*

   The research is focused on the implementation of a strong playing MCTS player. Thus, from the beginning (which will be the implementation of the game with its rules) we should take care that the structure of the program enables the MCTS to perform as many simulations as possible, because this directly leads to an increase of the playing strength. The first research question points to this problem.

2. *Which is the fastest way to check for a terminal position in TwixT?*

   The second research question deals with another important aspect, which is interrelated with the first one: Since TwixT is a connection game, each move may include the connection of some of the placed pegs. The game is over, when one player connects one side of the board with the opposite side. For MCTS it is a necessity that we find an easy way to detect this. Doing this iteratively through all of the pegs after each move might work, but is not the fastest approach to solve this problem.

3. *Which domain-independent enhancements of MCTS are useful for TwixT?*

   Since we want to implement an MCTS player, we have to think about domain-independent enhancements of the algorithm. Thus, the purpose of the third research question is to find those enhancements that improve the playing strength of the AI player.

4. *What kind of TwixT knowledge can significantly improve the MCTS program?*

   For the last research question, we add domain-dependent enhancements to the MCTS algorithm. Examples would be the modification of the selection, the modification of the playout and an earlier termination of the game.

After answering these four questions, we can use the gained knowledge to implement a strong MCTS player. It will be interesting to see if it is able to defeat other AIs that currently have been programmed. In addition, we might learn more about the game of TwixT as well as about the MCTS algorithm.

1.5 Thesis Outline

Chapter 2 describes the way how the TwixT rules were implemented in the MCTS program. This includes subtopics such as the relevancy of speed, connections between pegs and the detection of terminal positions. The first two research questions will be answered in this chapter.

Chapter 3 addresses the third research question. The development and implementation of a MCTS player is described in this chapter. It provides a brief description of the MCTS algorithm itself. Further, some domain-independent enhancements are described and discussed.

Chapter 4 discusses possibilities to include knowledge-based enhancements to the MCTS player and thereby answers the fourth research question. Different possibilities to calculate heuristic values are discussed. Afterwards, the weighting of these heuristic values as well as the heuristics in the final implementation are described.

Chapter 5 provides the conditions and especially results of the tests with different MCTS configurations. Our program TWIXTER is tested against the TwixT programs T1J and Moesker's program.

Chapter 6 is the last chapter and will contain the conclusions we can draw from the entire research. We revisit the initial problem statement and the research questions. Problems which occurred during the research are stated and discussed, possible solutions shown. Finally, an outlook on future research is given.

Chapter 2

Implementation of TwixT

This chapter describes the implementation of the TwixT rules in the computer program. The emphasis is put on the recognition of possible and impossible bridges after a move and on the detection of a terminal position.

Chapter contents: Implementation of TwixT – Relevancy of Speed, Past and Possible Moves, Pie Rule, Connections, Terminal Position Detection, Chapter Conclusions

2.1 Relevancy of Speed

When developing an AI for a game, it is always important to keep in mind the importance of speed. As was mentioned in the previous chapter, higher speed directly leads to better results in the game. This is important when implementing the search algorithm itself, but also when implementing the rules of the game. As the MCTS algorithm – which will be discussed in detail in Chapter 3 – is based on simulations of entire games, we have to make these simulations as fast as possible. When these games are played randomly, the “players” do not need much time to calculate their moves, so almost the whole amount of time will be used for the simulations. As we want to perform many simulations for every move, the code that runs a game and checks the rules is called often. Therefore, we should not just implement the rules, but focus on the implementation that promises the best performance and thus leads to the highest playing strength of the TwixT program.

2.2 Past and Possible Moves

Both human players and computer players have to check whether a move is legal. Computers need all legal moves when they try to find the best one. For TwixT, this check is quite easy. As described in the rules section, a move consists of placing a peg in an empty hole on the board and adding all bridges that are possible from this position. The information needed to represent a move is nothing more that the coordinates of the placed peg. A move is valid if the coordinates of the peg are valid. They could be invalid if either they are out of range (out of the board or in the enemy’s home) or this move has already been played. To check the latter condition, we need to keep in the memory which move have been played so far. It is quite easy to do that. We can simply store the x and y coordinate of every move in an array. Another possibility is the storage of the entire board, using a two-dimensional array. This would reduce the time for checking if a specific position is empty or occupied. However, the number of played moves will always be small in comparison to the number of possible moves. Thus, the storage of the coordinates is preferred to the two-dimensional array when calculating possible links (see below). A two-dimensional array is used as well, but only to decide whether the moves of the players are valid (every empty field on the board means a possible move).

2.3 Pie Rule

The pie rule was not implemented. Implementing this rule is not trivial, as it affects the complete playing mechanism that usually does not contain features like swapping sides. So it would require some effort to implement the pie rule in a way that it works properly. In addition, the pie rule is not interesting from the angle of an AI developer. It only affects the first moves but not the remaining part of the game. It is much easier to test the strength of the AI by playing a equal number of games as white and as black. So this is what we do in the experiments. Moreover, an opening book could handle the pie rule.

2.4 Connections

When a peg is placed, we have to check for bridges that can be built from this peg to the other existing pegs. As shown in Figure 2.1, there are nine different bridges that are blocked by one specific bridge.

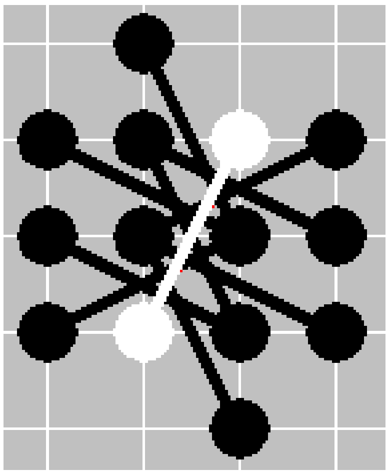

Figure 2.1: Bridges blocking each other

There are several approaches that could be used here. Most approaches check for the same conditions, but do this in a different order. The check requires several steps that have to be performed after each move:

- Find pegs that have the correct distance from the new pegs (on the opposite corner of one of the eight possible 2×3 rectangles).

  This step is the easiest one. When the new peg is set, all other pegs of the same color are checked for their distance. If the distance forbids a bridge to be placed between the new peg and the one that is currently checked, we do not have to do the other steps. This will happen most of the time, so we are saving much time by omitting the needless checks.

- Find own or enemy pegs that are in a position to block the possible new bridge.

  As a second step, we search for pegs that are "near" the new peg and the one that could be connected to it. In Figure 2.2, the pegs 5, 6, 7 and 8 are searched.

  This selection has an advantage: The pegs to be searched near the possible bridge, are selected in a way that, if none of these pegs nearby are found, the considered bridge is always possible. If we do not find any of the pegs nearby, we can immediately set the bridge and again avoid needless checks. Even if one or more of the nearby pegs are found, we save some time here, as, in the next step, we do only have to consider the pegs that have been found nearby.

- Check the bridges of the pegs found nearby.

  In the third step, we iterate over all bridges (of the correct color). We consider only that bridges that include one of the nearby pegs, because all other bridges are not relevant. This reduces the number

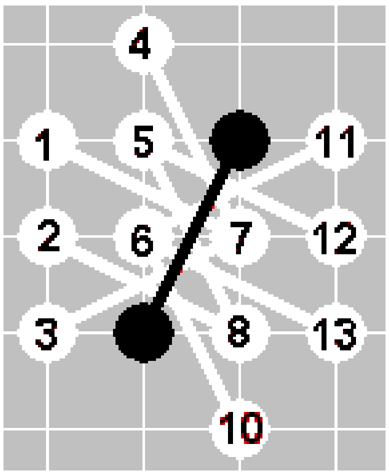

Figure 2.2: Pegs that can block a bridge

of checks at this step significantly, because the nearby peg can only be part of a limited number of bridges. For each bridge that we find at the nearby peg, we have to check in which direction it is leading. This is the final step, because it provides us with all the information needed. If there is no bridge between any of the pegs found in the second step and any of each one's corresponding pegs (corresponding: those pegs that block the potential new bridge if they are connected, e.g. 6–10, 6–12 and 6–9), we can set the considered bridge. If we find one of these connections, we cannot set the bridge.

Unfortunately, the last step is quite complex. But as one might see from the description, this is not a result of the approach to distinguish possible and impossible bridges. It is a result of the problem itself and of the high number of possibilities to block a certain bridge. This high number is a direct result of the rules. A similar check would be needed for any other implementations as well.

2.5 Terminal Position Detection

Implementing a check for a terminal position would be easy if there was no time constraint. After each move, we could check each player's pegs and bridges and step by step try to find a path from one side to the other. If we find a path, the game is over, if we cannot find such a path, there are some moves to be played. Unfortunately, we have a time constraint here, so we cannot take the easiest way for such a check. Instead, we should build up our information about the structure (the moves played so far and the existing bridges) incrementally on every step, so it is not required to reevaluate the whole board on every move. We have to store this information in a way that easily allows us to find out which pegs are connected with each other - directly or indirectly - and which are not. Further, we would like to know which pegs are connected with one of the home lines, which will enable us to detect a terminal position, as described below.

After some research, it was possible to implement a high-performing check. Each new peg gets an ID. There are three cases:

1. A peg in the upper or left line gets ID 0
2. A peg in the lower or right line gets ID 1
3. Any other peg gets an ID that has not yet been used

When a bridge is built (as a result from a new peg), we consider the IDs of the connected pegs. If they are equal, nothing has to be done (a loop (ring) may have been created), which will not occur often and does not affect the structure of the pegs). If they are not equal, we choose the smaller ID as the new ID for every peg that currently has a higher ID. The recalculation of the IDs has to be performed only once for each new bridge that has been placed and does require nothing more than a single loop over all pegs of the same color. After the recalculation, all newly connected pegs - directly or indirectly - have the same ID. Thus, we are creating nets of pegs incrementally with every move and have an easy way to find out if two pegs are connected or not - they are if and only if their IDs are equal.

Due to the fact that we always choose the lower of the two occurring IDs, we make sure that the IDs of pegs in the home lines are not changed. As a result, all pegs that are connected with the upper or left line have the ID 0, while all pegs connected with the lower or right line have the ID 1. Checking for a terminal position is quite easy now: If a bridge is built between two pegs with ID 0 and ID 1, the two home lines have been connected. In this case, the game is over and the current player has won.

This is a fast technique, because it makes well use of the information it has on every move. Thereby, it avoids recalculating the information which pegs are connected with each other and which are not. In addition, the memory needed for this check is small, because we only need one integer value per peg on the board. In the basic MCTS implementation on a 8×8 board and with 10 seconds per search, only 3.3% of the search time and only 4.8% of the time spent on simulating the games are used to check for terminal positions.

This approach has some disadvantages. First, it is difficult to include some kind of undo-mechanism, allowing to remove the last peg and restoring the previous state of the board. To make this possible, much more information would have to be stored. But in the MCTS framework, we do not need to undo moves during the move calculation (undo moves e.g. when a human player wants to change his last move will be possible as well, because we can regenerate the information from the order of moves that have been done, but it would be too slow to do this many times during the simulations).

A further problem is the recognition of draws. The described approach has no possibility to detect a draw, except for implementing a threshold that indicates that the board is full (and a draw is the only possible outcome left). But with other approaches, it would not be much easier to detect draws. Moesker (2009) decided to store two different networks for each player: One network represents all bridges that the player built, and the other network represents all bridges that are still possible to build in the future. When White does a move, one of his possible bridges becomes a factual bridge. When Black does a move, all of White's bridges, that are not possible any more, are removed from his network of possible bridges. These are all bridges that contain the position that Black has just occupied, and all bridges that are crossed by any new bridges of Black (if there are any). With this mechanism, a draw can be detected quite easy: If neither White nor Black have a path between their home lines through their network of possible links, the game cannot be won by anyone and is a draw. Unfortunately, calculating and storing all this information requires a high amount of computation time and memory. Moreover, the implementation of this mechanism is not trivial.

Though the draw detection is suboptimal and loses some time (trying to finish games that will end as a draw anyway), it is still reasonable to use this approach. Humans are able to see a draw when playing against the AI. Two AIs playing against each other do not have a problem here because they can play until the board is full and they cannot move anymore, detecting the draw this way.

2.6 Chapter Conclusions

The implementation of the rules are more complicated than one would have thought beforehand, especially with respect to the time constraint.

Storing past and finding possible future moves is not a complex task. The chosen data structure seems redundant, because both a list of moves and a two-dimensional array are used (where the latter could be generated out of the former on demand), but this is no problem. The amount of memory space lost due to this redundancy is small. On the opposite, the data structure enables us to do all necessary operations quickly.

Distinguishing possible and impossible bridges between pegs turned out to be more complicated. The high number of possibilities to block a specific bridge leads to complicated check which could not be avoided. However, the check was designed in a way that most of the conditions are not checked a lot. The easiest checks are done first, so that the complex checks are often not necessary. This resulted in non-elegant source code, but keeps the speed high.

The detection of terminal positions could have been solved in a theoretical and complicated way (Moesker, 2009). But another approach has been chosen. It is practice-orientated and avoids unnecessary computations. Thus, it saves much computation time and memory space. Another advantage of this approach is that it was easy to implement.

In total, we have solved the problem of creating a high-performing algorithm that checks the game rules. It required a thorough analysis, but it enables AI players developed later to do as much simulations in a fixed amount of time as the TwixT rules allow it.

Chapter 3

Monte-Carlo Tree Search in TwixT

In this chapter, the basic idea of MCTS with emphasis on the implementation in the program is described. In addition, we consider some domain-independent enhancements of MCTS, their implementation and usefulness.

3.1 Data Structure

Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári (2006), Coulom (2007)) is a best-first search based on Monte-Carlo simulations (Chaslot *et al.*, 2008). Further, some other, fixed information is given, such as the board size and the time available for the calculation. But before we address the algorithm itself, we have to consider the information that we will work with in order to find a proper data structure to store this information.

Each node in MCTS has exactly one parent and a variable number of children, with the total number of possible moves as a upper bound. To represent the whole tree and to have access to all nodes, it is sufficient when the MCTS algorithm has access to the root node. From there, we can reach every other node by recursively exploring the children.

In addition to the tree information about parents and children, we need to store the semantic information in the nodes. Each nodes represents one possible move, which consists of two coordinates x and y. These coordinates uniquely identify the nodes. Further, we can use these information to reconstruct the moves of the players, leading to the given position.

We also have to provide the possibility to perform the backpropagation process, described in the following section. Therefore, each node needs counter variables: tracking the number of games, the number of wins of the current player, the number of losses and the number of draws (it is sufficient to store three of these four values, as the last one can be reconstructed of the three values and we do not want to store more data than necessary).

3.2 Implementing the Monte-Carlo Tree Search

The entire MCTS is controlled and directed by a central loop. It runs until the time is over, and it calls all steps required to perform at a specific situation. Before the loop itself is started, the (empty) root node is created and stored. After that, each iteration can be performed in the same way. First, the MCTS selection is performed. It returns a leaf node of the tree as well as the move that should be performed next. The expansion method uses this information and adds a number of new nodes to the selected leaf node. Then, the MCTS playout plays a simulated game. The playout returns a value that indicates the result of the simulated game - either a win for one of the players or a draw. Depending on this, the backpropagation is performed, that evaluates the results and updates all values from the current node to the root.

The following sections describe the basic implementation of each one of the MCTS steps, (1) Selection, (2) Expansion, (3) Playout and (4) Backpropagation. The four steps are shown in Figure 3.1 as well (Chaslot *et al.*, 2008).

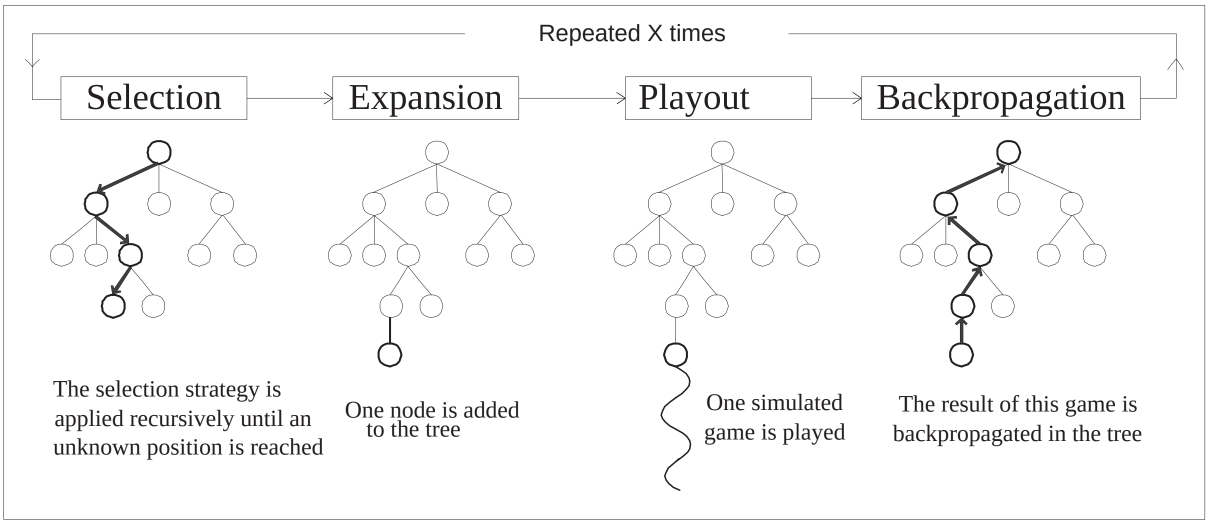
Repeated X times
Selection
Expansion
Playout
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One node is added to the tree
One simulated game is played
The result of this game is backpropagated in the tree

Figure 3.1: The four MCTS steps

3.2.1 Selection

Each run of the main loop begins with the selection of a node. The following three steps – Expansion, Playout, and Backpropagation – depend on the node we choose here. In the selection process, we have to find a balance between two major aspects: exploitation and exploration.

For exploitation, we try to get more information about those nodes that are likely being chosen. Collecting information about the difference between two nodes that have a low score is useless, because we do not select one of the nodes anyway. But for two nodes, that both have high scores, we would like to have more information. These nodes are likely to be chosen, and more information on these nodes can have an effect on the final decision.

However, we should not concentrate too much on promising nodes and totally forget the other nodes. Maybe some of the other nodes are not looking promising because they have not been played enough. As there is a strong dependency on random values, a good node's quality is revealed only after a number of games played through it. So we have to combine the concept of exploitation with exploration. In exploration, those nodes are preferred which have not been selected often. This enables that all nodes are taken into account. After a number of games are played through a node, we can better decide whether it is good one or not and whether we should spend more time on it or not.

The exploration aspect has a higher variety of options (Chaslot *et al.*, 2007). A common feature of all implementations of the exploration is that the value of a node decreases as the number of games played through it increases. This is necessary because the more simulations that have been performed on the current node, the more we can rely on the result and the less we have to explore it.

It is not possible to find the "best" trade-off between exploration and exploitation. Each game has its own rules and characteristics, influencing the optimal trade-off. Further, the optimal way of combining the two aspects depends on the implementation of the game rules and especially on how realistic the simulations are. If the simulations are performed randomly, the information gain of each simulation is low. In that case, we should put more emphasis on the exploration, as we cannot rely on the simulation results. If the simulations are quite realistic, we will not need that much exploration and can concentrate on the exploitation. We can rely on the information that the simulations brought forth.

During the research, we add different features to MCTS. As we do that, the parameters for the trade-off are affected. Thus, we would have to tune the trade-off after every major step of the research. To avoid that, we implement a reasonable trade-off and keep it for a while. As we get nearer to the final product, we can fine-tune the parameters to improve the final playing strength.

The selection process is as follows. It is a path that starts at the root node and ends at a leaf node. In every step, a value for each of the children of current node is calculated. This value depends both on the exploitation and the exploration. Usually, the UCT formula (Kocsis and Szepesvári, 2006) is applied to calculate these values, which is also used in our program. The UCT formula is used to choose the node k to be selected:

$$k \in argmax_{i \in I} \left(v_i + c \times \sqrt{\frac{ln(n_p)}{n_i}} \right) \tag{3.1}$$

where v_i is win proportion of the node n_i, c is the variable that handles the trade-off between exploration and exploitation (experimentally set to 2.5), n_i is the visit count of the current node and n_p is the visit count of the parent node of n_i. After the values of all children have been calculated, the one with the highest score is chosen. In addition, a small random value is added to each node's value to make sure that the selection does not bias the first added child (or a similar behavior that is not intended).

The described selection function assumes that every child node has been visited at least once (because the number of games played on the child appears in the denominator, which is not allowed to be zero). When a certain node has not been visited so far, something different has to be done. The value of this node could be estimated, but these values are difficult to estimate. Preferably, the value of nodes that have not been played so far should be set to a value that other nodes cannot reach. As the values of not visited nodes are always higher than those of the visited ones, each node is chosen at least once before the better ones are preferred.

Finally, UCT does not produce good results at a node that has not been visited frequently. As long as the threshold t is not reached, the next node is simply selected as done in the playout strategy (Coulom, 2007).

3.2.2 Expansion

After the selection is finished, we have to expand the tree. The selection step chooses a node that has to be added to the tree. The most simple version of the expansion algorithm adds only the first encountered position in the simulation (Coulom, 2007). In the TwixT program, we use this version of the expansion step. But there are other versions available. Chaslot *et al.* (2007) proposed to add the siblings as well (if a certain threshold is reached).

3.2.3 Playout

In the playout, we have to simulate the rest of the game (starting from the newly added node in the tree). The basic playout would be an all-random strategy that draws uniformly a move. This may produce unrealistic results, but is easy to implement. In addition, it does not consume much time and can be performed quite often in a fixed amount of time. The simulation may be bad, but as we get more and more simulations, we may obtain an estimate of a higher quality.

A playout based on heuristic knowledge would be much better, even if it is a simple one. Any reasonable simulation strategy improves the quality of a simulation. Each simulation is more realistic and has a higher information gain. But, if the simulations become too knowledge intensive, only a small number of simulations can be performed in a fixed amount of time. This could reduce the quality of the final decision. In Chapter 4 we discuss which kind of knowledge can be added to the simulation strategy and how this is done. As long as no knowledge is used, the games will be simulated completely randomly.

3.2.4 Backpropagation

To affect the upcoming simulations of MCTS, the value has to be backpropagated. When a simulation is done, the value of the node where the simulation started is updated. Afterwards, the same is done iteratively for all other nodes between the playout node and the root node (with the side effect that the root node is updated after every simulation).

Updating a node means to recalculate the number of moves played through it and the number of wins that returned from it. As experience shows, a simple but good strategy is the number of wins divided by the number of games played (Chaslot *et al.*, 2007). Other strategies have been proposed, but none proved to work better that the basic version.

An issue that could be handled differently is the counting of draws: Draws can be completely ignored (neither increasing the numerator nor the denominator). They could also be counted as losses for both players (increasing only the denominator) or somewhere in between (increasing the denominator by one and the numerator by a value between 0 and 1). Regardless of how draws are handled, a move that always lead to a win will get value 1 (and will be chosen often, as other values cannot have higher values that 1), a move that always leads to a loss will get value 0 (and will be chosen rarely, as other nodes will almost always get values higher that 0). All other moves are somewhere in between. For our MCTS program, we have chosen the basic backpropagation strategy with draws counted as a half win for both players.

After the backpropagation, we got a slightly modified tree. We start a new MCTS run (as long as the time is not over), thereby using the modified data and thus incrementally increasing the quality of the search.

3.3 Domain-Independent Enhancements

The MCTS algorithm described in the previous section is the basis for all other work on the search. Knowledge about the domain can increase the quality of the search. But there are also enhancements available that do not require the use of knowledge about the domain. These will be described in the following subsections.

3.3.1 RAVE and Progressive History

A problem of MCTS is the fact that it takes a long time to find the correct "direction" of the search. It takes quite a lot of simulations in several directions until the selection strategy starts to choose promising moves. This problem can be encountered with a concept called Rapid Action Value Estimate (RAVE) (Gelly and Silver, 2007).

The idea of RAVE is the following: beside the scores that are tracked for each node, another score is tracked for each move that is possible in the game. For TwixT, we view all moves equal that begin with the same peg to be set. If during the backtracking a node is reached that represents this move, we will not only modify the node's score, but also the score of the move itself. To keep track of these scores, we need two tables for each player: one for the number of times a move has been played (separated into one score for White and Black) and one for the number of wins that this move led to (again separated for White and Black).

Calling RAVE "domain-independent" is not fully correct as it makes an assumption of the underlying domain. It assumes the following: Moves that are a good choice right now, will also be a good choice a (small) number of moves later. Further, a move that is a good reaction on a certain opponent move will also be a good reaction on other possible opponent moves. These assumptions are not correct for every possible game situation in TwixT. But they are roughly correct, and that is the information we want to use.

For the standard RAVE, these tables have to be stored in every node and apply to all playouts that started below this node. But in TwixT , using all these tables would take too much space. Instead, only two global tables are used for each player. This approach decreases the quality of the RAVE values, but it is much easier to compute (and to implement) and requires less memory. Further, the quality of the tables of nodes in the lower levels of the tree is not high. These nodes are rarely visited, so the RAVE values are not reliable. So the decrease in quality will not be severe.

For our implementation of RAVE, we modified the selection formula 3.1. Instead of the standard UCT, we use:

$$k \in argmax_{i \in I} \left(v_i + c \times \sqrt{\frac{ln(n_p)}{n_i}} + \frac{w \times h_i}{n_i + 1} \right) \tag{3.2}$$

where h_i is the RAVE value based on move i and w is the weight of the RAVE feature (which we have to tune). For those moves that have not been played yet ($n_i = 0$), v_i is assumed to have the maximum score and the middle term is not taken into account. As this variant does not follow the standard RAVE, it is called progressive history (PH) (Nijssen and Winands, 2010).

There is a further variant of this formula (Nijssen and Winands, 2010). As long as a certain node continues to produce good results, we do not want to decrease the PH value. We only want to decrease the PH value when the simulations on the node produce low scores. Thus, we can also use the following formula:

$$k \in argmax_{i \in I} \left(v_i + c \times \sqrt{\frac{ln(n_p)}{n_i}} + \frac{w \times h_i}{n_i - s_i + 1} \right) \tag{3.3}$$

where s_i is the score of node i (somewhere between 0 and n_i). All other values are the same as in Figure 3.2.

3.3.2 MCTS-Solver

Another enhancement that does not depend on the game domain is MCTS-Solver (Winands, Björnsson, and Saito, 2008). In addition to the estimation that is done for every node in the tree, the MCTS-Solver tries to find final, proved values. Its advantage is that it detects moves that will lead to a terminal position and can either choose these moves or try to avoid them - depending on whom the winning move belongs to.

Implementing the solving algorithm in a standard MCTS engine, two parts of the search have to be modified. First, the playout: Before the usual playout is performed from a certain position, the solving algorithm tries all responses to the last move once. It does not perform a complete playout for each of these responses, but only checks whether they end the game or not. So it does exactly one additional step (the corresponding node has not yet been created) and stops the game afterwards. In TwixT, each terminal position is a win for the one who played last (except for draws). So when we encounter a terminal position, we know it is a win for the current player and a loss for the other one.

The second MCTS step to be modified is the backpropagation. The modified playout gives us a certain piece of information: Either, the first move in the playout enables the opponent to end the game with a win, or it does not. If the opponent has now the chance to win, we can mark the current node as "solved". It always enables the opponent to win, and thus we will not play it anymore. Any other node cannot be worse than this one.

Now we start the modified backpropagation. If the current node was solved as a win, we can immediately mark its parent as a loss. We would always choose the winning move from this position, so it will never produce any other result than a loss. Then we continue with the backpropagation one level above. If the current node was solved as a loss, we cannot immediately mark its parent as a win. We can only do that if all siblings of the current node have also been proven as losses. In all other cases, we cannot mark the parent. As soon as we cannot mark a node or if we cannot mark anyone at all, we continue with the usual backpropagation.

Computing this information can be expensive, but it can save quite some time near the end of a game when many terminal positions occur in the tree. To decrease the effort without losing too much of the information gain, the solving algorithm had to be modified. In TwixT, there are many moves that are unlikely to be terminal moves. In most of the cases, the game is won by a move that is placed in one of the home lines. Other moves are also possible (if there are already pegs in the home line), but this occurs rarely. So we decided to exclude all moves that are not in the home lines. Only two lines per player are checked for terminal moves, which severely reduces the amount of computation to perform. Chapter 5 shows the results of the solving algorithm.

3.3.3 Reusing the MCTS Tree

Different from other search techniques, such as alpha-beta, MCTS builds up the tree incrementally. Thereby, each run of the main MCTS loop (see Section 3.2) produces almost the same amount of information. It gives an estimation of the node's value. This estimation does only depend on the board situation that the corresponding node represents. If that node was in a different tree, but had the same sequence of moves leading to it, it would have exactly the same value. This may sound trivial, but it is an important piece of information that we can use.

When the AI has built the MCTS tree and the time is over, it chooses one of the children of the root node. Usually, it is the node with the highest visit count – sometimes the visit count is even the criteria which node is chosen. Thus, a relatively high percentage of all simulations have been played through this node. After choosing it, the opponent will make his move. As MCTS also predicts the moves after the current move, the opponent will often also choose a move that has been visited quite a number of times. This leads to the conclusion that a high number of moves have been played through this node that will be the root node when calculating the next move.

It would be a waste of time to start over again when we calculate our next move. Therefore, we should start the calculation with the former subtree, whose root represents the updated, current position. So, in the beginning of the calculation (before any kind of search is performed), we have a look to the tree that is the result of the last calculation. We have to make sure that we are still in the same game, and have to determine which part of the old tree fits the new position. If we find a position in the old tree that corresponds to the root position of the tree we would create in this calculation, we use the old tree.

A nice property is that the part of the old tree that can be reused increases the general quality of the search. This feature is quite easy to implement and especially needs a low amount of calculation. So even if the gain might not be high, we should try to use it.

3.3.4 Backpropagation With a Discount Rate

The first simulations of each MCTS run are played almost randomly. The algorithm has not yet any information about good or bad moves. After some simulations, this information is gained and gets more and more reliable. The second half of the simulations on a certain node is more realistic and thus more reliable that the first half. So, it may increase the playing strength if the later simulations are higher weighted than the earlier simulations.

This can be shown by an example: There are two favorite moves in the search that seem to have a similar value. The first move gave 80% wins after 150 simulations, the second gave 70% wins after 100 simulations. We would choose the first move due to its higher score and a higher number of simulations. But this might not always be the best choice. The first move may have won 95 of the first 100 simulations, but only 25 of the last 50 simulations (120 in total, which equals 80%). As the later simulations are more realistic, this might indicate that the 50% win rate of the last 50 simulations is a better estimation of the move than the 95% win rate of the first 100 simulations or the combined 80% win rate. The low rate in the last 50 simulations might also indicate that the next simulations will produce similar results, further decreasing the move's value. Further, the move that scored 70 of 100 simulations might be underestimated. Maybe it won 25 of the first 50 simulations, but 45 of the last 50 simulations. Thus, the next simulations will also produce a rate near 90% and the value of the move will increase in the future. So, we lose information if we simply take the win rate of the move to estimate the move's strength.

A simple and easy-to-implement approach to solve this problem is to use a discount rate. Each time a node is updated during the backpropagation, we also modify the score that has been achieved so far. This is done via a discount factor d with $0 < d < 1$. Before the counter and the score of the node are updated with the new value, we multiply both the score and the counter by d. As this is done at each update, the information of the first simulations is decreased often as more and more simulations are performed. The later simulations are not multiplied by d often, so the information they produced is taken into account with a higher weight.

The value of the discount rate has to be tuned. If it is too close to 1, the difference to the standard backpropagation is too small to be recognized. But if it is too close to 0, we would discard useful information. The first simulations at a node are not as important as the last simulations, but they still give valuable information. Even with a value of $d = 0.8$, a simulation score will after 10 other simulations drop to $0.8^{10} = 0.107 = 10.7\%$ of the initial score. As there will be thousands of simulations, the optimal value of d might be somewhere above 0.9.

3.4 Chapter Conclusions

MCTS is a well-researched approach in Game AI. It provides a smart search algorithm that concentrates on promising moves. It is easy to find a trade-off between exploitation and exploration by simply running several tests with different parameters. In addition to this advantage, it is also possible to enhance the given algorithm by other concepts, both with and without knowledge about TwixT.

Implementing MCTS in TwixT is not a hard problem in comparison to other games such as chess and checkers. It is easy to get a list of all possible moves which are equal to every vacant position on the board. It is also an advantage that every TwixT game ends at a certain point – either one of the players won or the board is completely filled with pegs. Other games have the possibility that a playout lasts forever, requiring the programmer to define a stop condition.

Further, the MCTS framework provides us with the possibility to add knowledge to the AI. The next chapter describes which knowledge we can use and how to apply it.

Chapter 4

Using Knowledge About TwixT

In this chapter, we discuss how to use knowledge about TwixT for our program. It is described where and how the knowledge is to be used.

Chapter contents: Using Knowledge About TwixT – Applying Knowledge in MCTS, Implementing and Weighting Basic Heuristics, Other Features, Chapter Conclusions

4.1 Applying Knowledge in MCTS

In general, there are two applications of knowledge in the framework of MCTS: (1) Knowledge can be used in the selection step. (2) Knowledge can be used in the playout step.

4.1.1 Knowledge in the Selection

As described in Section 3.2.1, the selection step in MCTS encounters some problems when a certain node has not yet been visited often or if a child node of it has not been visited at all. In this case, we suggested a more or less random distribution. As soon as we use domain knowledge, we do not have to use the uniform probability distribution anymore. Even if the knowledge is poor, it will be better than guessing the values of the nodes. Thus, we can use the knowledge here.

The selection step is performed at every node of the path going from the root node to the newly added leaf node. This happens quite often, but still not often in comparison to the playout moves. Thus, we could implement a quite complex function that returns a probability for every possible move. But there are several reasons – besides the time issue – to keep the knowledge simple. Simpler knowledge can be implemented much faster and will be less prone to errors. It is easier to find good parameters and the estimates for them will be more reliable. Simpler knowledge can be adapted to knew circumstances easier as it does not depend on the size of the board, the strength of the opponent, and other circumstances that may change. In the simplest case, we use the same heuristic as in the playout strategy. This reduces the work that has to be done on implementing and testing the heuristics.

4.1.2 Knowledge in the Playout

The most important place to use knowledge in MCTS is the playout. As already mentioned in Section 3.2.3, the basic version of the playout chooses a random move of all possible moves. It distributes the probabilities of all moves uniformly. This leads to the first problem, playouts that are unrealistic and unreliable. In the case of TwixT (and probably in many other games as well), we encounter the second problem with the basic playout strategy: The simulated games are quite long. A random player does not know the concept of bridges, even not the aim of the game. A match that consists of many moves will also take too much time to be simulated. Thus, we get a relatively low number of simulations with a relatively low reliability.

These problems can be solved by using a smart simulation strategy. The aim of the strategy is making the simulations more realistic and shorter. More realistic simulations directly increase the quality of the decisions that are made based on them. Shorter simulations simply allow us to do more simulations in the same amount of time, further increasing the quality of the decisions. Generally, a simulation strategy provides a value for every possible move (depending on the current board state). To simulate a move in the playout, we use these values as a probability distribution. A move that has the double value of another move has also the double probability to be chosen (we do not automatically choose the move with the highest score).

A high emphasis has to be put on the CPU time that is needed for each move in the playout. An example with realistic values: Assume we play on the 12×12 board, having 100 moves available (on average). We want to simulate 1,000 games per second. Each of these games has a length of 100 moves. So, in every second we need to know $100 \times 1,000 \times 100 = 10,000,000$ different probabilities. Each of them stand for the probability of a certain move in a certain situation in a certain simulation. As we can see, it is worth the effort to implement the method in a way that works really fast.

As many heuristic features apply for a group of moves instead of a single move, it would be a waste of time to calculate the new probability for each of the moves. Instead, we calculate the probabilities of all possible moves at once and use the probability for each of the moves. For example, when one line of pegs should have another probability than other pegs, we calculate the new probability once and apply it to all pegs in that line (applying can mean to add it, to multiply by it, or another operation). With this simplification, we do not have to calculate the probabilities of the moves separately, thereby severely reducing the calculation time.

We can further simplify the calculation of the probabilities. After n moves of a simulation, we have a certain probability distribution. We use this distribution to choose the next move of the simulation. Now we need the probability distribution after $n+1$ moves. It will be similar to the distribution after n moves. So it would save time to use the old values again instead of recalculating them. Then we only need to apply the domain knowledge to the last move and modify some of the probabilities. So the impact of each move of the simulation on the probabilities of the remaining part of the game is calculated only once, straight after the move has been chosen. We also do not have to store the old values because they are not needed anymore.

4.2 Implementing and Weighting Basic Heuristics

This section discusses some of the possible effects that a move in a simulation has on the probability distribution of the remaining part of the simulation. Generally, every piece of knowledge results in different weights for the different remaining moves. Every probability in the distribution is then multiplied by the corresponding weight. The factor of many possible moves will be 1, so only some of the probabilities are affected (and we save a lot of time). After the modified probabilities have been computed, we calculate the values of the next piece of knowledge.

4.2.1 Adding a Bridge

As the game can only be won through a chain of bridges, a simple idea is to increase the value of every move that can be connected to the recently added peg. When placing a move at (x, y), we multiply the values of $(x-1, y-2)$, $(x-2, y-1)$ etc. with the weight w_{Bridge}. When several pegs are connectible to a certain peg, its score will also be multiplied several times. We do not check for bridges that block the new bridge, because this check consumes a lot of time. It would not increase the quality of the probability distribution.

As we want not only realistic, but also short simulations, we prefer a certain kind of bridges. Two pegs that are connected by a bridge have either a x-difference of one and a y-difference of two or vice versa. We call these cases y-directed and x-directed, respectively. White, playing from left to right, will have a higher win probability if he prefers x-directed bridges, while Black should prefer y-directed bridges to connect his home lines as fast as possible. So it may be useful to use different weights for these different types of bridges. All moves that are connectible to the new move with the "good" type of bridge will get multiplied with a higher weight, the moves that are connectible with the "bad" type of bridges with a lower weight.

Preferring holes that may result in one or more bridges may cause a problem: When a few moves are played in a certain area, holes in that area may get high values to be chosen as the next moves. After a several number of moves, many holes in this area are occupied by pegs, which is not realistic. When a new move is added, we only increase the probability of its neighbors if the new move has not a high number of neighbors yet. So a new move that is already connected to n or more pegs does not increase the probability of its neighbors. But if the new move creates no or only a small number of bridges, the neighbors' probabilities are updated. Further, the weight depends on the number of bridges to the new peg: The lower the number of the bridges, the higher the weight with which the neighbors are multiplied.

We have to take another characteristic into account: Building bridges is not equally important in all phases of the game. Maybe it is more reasonable to play almost randomly in the beginning and start connecting the set pegs later on. In this case, we should increase the weight for possible bridges during each simulations. But it could also be vice versa: Maybe it is useful to build bridges in the beginning and to play more randomly near the end. In this case, the weight for the bridges should be decreased after some simulated moves. So we should test this feature in both directions: Increasing the bridge weight during the simulations as well as decreasing it. We also have to test different parameters for the modification of the bridge weight.

As we want to test both an increasing and a decreasing weight during the simulation. Therefore, we choose the following formula for the weights:

$$w_{Bridge} = 2 \times \frac{moves_{max} + w_{Count} \times c_{curr}}{moves_{max}} \tag{4.1}$$

where $moves_{max}$ is the maximum number of moves per simulation, w_{Count} is a parameter that we have to tune, and c_{curr} is the current move count of the running simulation. As we see, a higher move count leads to a higher weight for the bridge, up to the double weight for $c_{curr} = moves_{max}$ (but as the games that do not end in a draw usually end earlier, we do not get such large weights here). The variant for decreasing weights is similar:

$$w_{Bridge} = 2 \times \frac{moves_{max}}{moves_{max} + w_{Count} \times c_{curr}} \tag{4.2}$$

where all symbols are equal to the first variant. As we see here, the weight decreases with a growing c_{curr}, with half of the weight being the lower bound.

4.2.2 Prefer Rarely Used Lines

As described in the previous Section 4.2.1, we may have the problem of preferring moves in an area that already contains many moves. To avoid this behavior and make the simulations more realistic, we want to prefer moves in lines (rows and columns) that do not contain many moves so far. The fewer pegs a line contains, the higher the probability of a move that line is. Therefore, each move's probability is divided by the percentage of empty pegs in its line. As it is hard to include a weight in this feature, we will simply test it with the described implementation, without using weights.

4.3 Other Features

Besides the modification of the probability distribution, some special features of TwixT can be used. These are described in the following sections.

4.3.1 Game Termination

When a game ends in a draw, this is often evident a long time before the game terminates. The difference between the first position where no player can win anymore and the termination (due to a full board) is smaller when playing randomly or with little knowledge, but there is still a possibility to save computation time. A simple way to modify the simulation strategy is to limit the number of moves. When this number of moves is reached and no one has won yet, the game is considered a draw. We may end some games that would have not been a draw, but there is a reason to tolerate this. The major part of the terminated games would really have been a draw.

The implementation is quite simple. The TwixT implementation (Section 2.5) already contains a threshold after which the game is terminated (depending on the board size). If this threshold is reached, the game is counted as a draw. If we decrease the threshold, the games are ended earlier, but number of draws might increase, but the total number of games performed (and hopefully the number of non-draws) increases.

4.3.2 Excluding Home Lines

As showed in Section 1.2.1 (see also Figure 1.4), there are not many positions where a player already connected the lines next to his home lines, but is not able to connect the home lines themselves. When a player reached a line next to one of his home lines, the opponent is not able to block him any more. So in most of the cases, it is a wasted move to place a peg in one of the home lines in the early part of the game. This should only be done when the lines next to the home lines have been connected and the home lines are needed to finish the game. This is an important piece of information that we want to include in the simulation strategy.

To implement this feature, we modify the probability distribution if a random value is below a certain threshold. This is done according to the following formula:

$$v_{Exlude} = rand(0,1) * w_{Exclude} - c_{curr} \tag{4.3}$$

where $rand(0,1)$ is a random value between 0 and 1, w_{Exlude} is the weight of this feature (which has to be tuned) and c_{curr} is the move count of the running simulation. v_{Exlude} is the value that indicates which of the distributions has to be used: if $v_{Exclude} < 0$, the home lines are allowed, otherwise they are not.

As playing in the home lines almost never contributes to the decision of the game, we can go even further in order to exclude them in the simulations. When a player connected the lines next to his home lines, he will almost always win the game. So, from the position in Figure 4.1 to the position in Figure 4.2, we gain almost no new information, but we still have to perform several moves in the simulation. We can save the amount of time by considering a position as in Figure 4.1 already as terminal (in this case, as a win for White).

The implementation of this termination condition is not difficult. We basically use the same terminal position detection as described in Section 2.5, that assigns IDs to every net of pegs, with special IDs for those connected to the home lines (0 for the upper and left row and 1 for the lower and right row). The only difference is that all pegs next to the home lines are viewed as already connected to them. So they have as default the IDs 0 and 1, respectively. Then the games will end earlier because we do not have to place pegs in the home lines. With this behavior, we can exclude the home lines completely from the simulations.

One special case has to be taken into account. When the current game is in a position as in Figure 4.1, the search should be able to find the winning move. If the games are terminated earlier, all moves would get the value 1, because all simulations end in a win. But then there would not be any difference between the moves that will end the game soon and all other moves. So the program will play completely random then and the last two moves are only found by chance. So we have to detect these kind of position and turn off the earlier termination in these cases. To do this, we examine the board position right in the beginning of the search. If we find that the game is already over – according to the earlier termination – we do not use the earlier termination in the search. This holds for both the usual search and the search with the MCTS-Solver.

4.3.3 Weighting Simulations by Length

Generally, all simulations give us a similar amount of new information. But if we were able to separate good and bad simulations (or, realistic and unrealistic ones, respectively), we could put more trust in the good simulations and less trust in bad ones. So, the estimation of a move's value would be closer to its actual value. One idea to realize this is to modify the backpropagation strategy. Instead of backpropagating the game result -1, 0 or 1, we backpropagate another value. This value should be closer to 0 for unrealistic simulations and further away from 0 for more realistic simulations.

To distinguish realistic from unrealistic simulations, we chose the game length as an indicator. We use a simple rule: The shorter the game, the more realistic it has been. So long simulations get a value close to 0, while short simulations get a value that is farther away from 0. This value is calculated by the following formula:

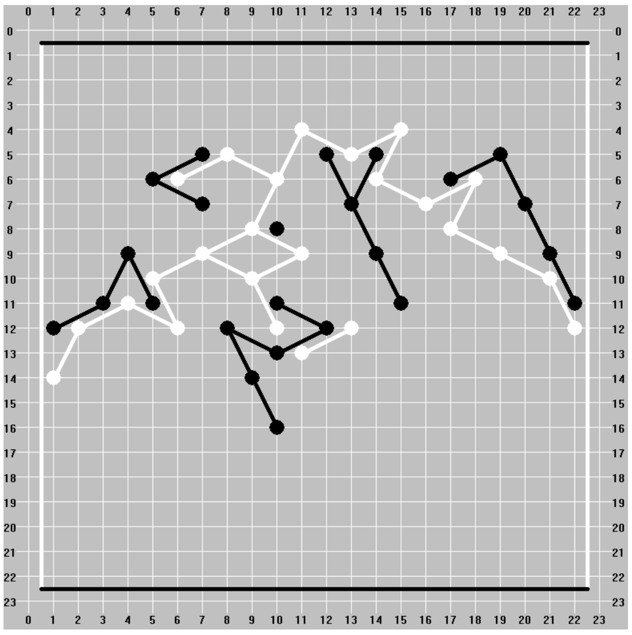
Figure 4.1: White connected the lines next to the homes.

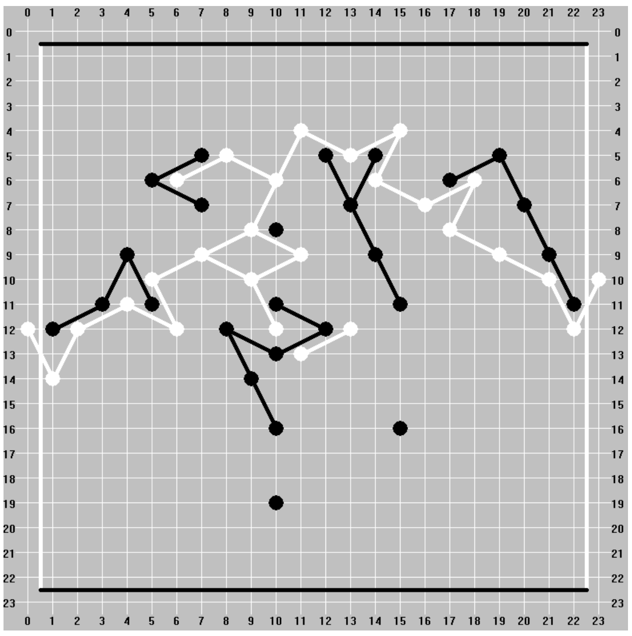
Figure 4.2: Terminal position, White wins.

$$v_s = v \times (1 - \frac{w_{Length}}{2} + w_{Length} \times \frac{moves_{max} - moves_s}{moves_{max}}) \tag{4.4}$$

where v_s is the backpropagated value, v is the result of the simulation, w_{Length} is the weight of this feature, $moves_{max}$ the maximum number of moves per simulation and $moves_s$ the number of moves in the current simulation.

As we can see, if $v = 0$, the same value is backpropagated. The score of a won game ($v = 1$) is between $1 - \frac{w_{Length}}{2}$ and $1 + \frac{w_{Length}}{2}$. When the game was short, the value is near the upper bound, if the game was long, the value is near the lower bound. If we want to include this feature in the final program, we also have to approximate the optimal weight w_{Length}.

4.4 Chapter Conclusions

There is a broad variety of possibilities to use knowledge in the MCTS algorithm. The most general approach to use knowledge is to modify the playout (and the selection strategy, as long as the MCTS threshold is not reached). In the playout, we have a probability distribution over all possible moves. The initial state is the uniform distribution, but by adding several features the distribution depends more strongly on the past moves of the current simulation. As there are many simulations, we have to ensure that the calculation of the probabilities is fast and does not decrease the number of simulations too much. Otherwise, the increase in the total AI strength would be low, or we would even make it weaker.

Besides the playout (and the selection before reaching the MCTS threshold), there are other possibilities to use knowledge. Most of these have to do with the termination of the game. A first idea that should be tested is to terminate the games after a certain number of moves. This number can be lower than the total possible number of games, so the games become shorter. This enables us to run more simulations, while a higher percentage of the simulations may end in a draw. Playing every game until the end might not be the optimal strategy.

Another idea that has to do with the playout is the exclusion of the home lines. This utilizes a feature that might be found in few other games. Some moves before the actual end of the game, we know the winner almost for sure. This can be used as a trigger to end the game and consider it as a win for the player who has almost won the game. Unfortunately, it will be hard to generalize this idea in order to use it in other games as well. But this should not hold us back from using it for TwixT.

Chapter 5

Experiment Results

Different features and settings have been compared by running many experiments. This chapter describes the experiments and their results. It also includes a description of the conditions of the experiments.

5.1 Conditions

When implementing an AI for a game, we generally have to run test games to distinguish good from bad features. Maybe a certain feature is a nice idea, but the implementation in a certain game is hard and it costs a lot of computation time to calculate the feature. Then an AI that uses the feature is playing weaker than another AI that does not use the feature (assuming that the AIs are identically except for that feature). So we cannot know in advance whether a feature will improve the strength of our program or not. We need to perform test games in order to estimate the playing strength of each AI.

5.1.1 Number of Test Games

Usually, an enhancement in the AI will not make it win 90% of the games against an AI without this enhancement. Often, only 55 to 60% of the games can be won. So we need to play as much test games as needed to recognize differences of 5%. In order to do that we decided to run 200 test games for each comparison between two AIs. A result 44% : 56% points is a significant difference in 95% of the cases. If the difference is smaller, we have to play more test games to be sure that it is significant.

5.1.2 Board Size

An important feature of the MCTS algorithm is the distribution of computation time. More promising nodes are visited more often than less promising nodes. This effect only occurs after a certain number of simulations that have been performed in the calculation. In the beginning, MCTS has no advantage in comparison with a greedy player (that simply distributes the calculation time uniformly and chooses the move with the highest win percentage in the simulations). If we want to examine the MCTS algorithm including enhancements, we have to choose the experimental conditions in a way that the effects of MCTS will be visible.

The original TwixT is played on a 24×24 board. As described in Section 1.2.2, this results in a high number of moves per game. If we assume an average branching factor of 452 (as claimed by Moesker (2009)), only $\frac{1}{452}$ of all simulations are played through each child of the root node (average). Through each node one level below, only $\frac{1}{452}^2 = \frac{1}{204{,}304}$ of all simulations are played. Therefore, we have to perform 204,304 simulations if we want to reach every node two levels below the root once. As long as we do not reach an order as high as 10^4 games per search, MCTS will not be a reasonable approach.

The first implemented setup, the greedy player, can simulate 517 games per second on this board size. When the MCTS algorithm is implemented, the value even drops to 475 games per second (due to some extra computation). Thus, the greedy player would need 395 seconds to reach the mentioned number of simulations. But more than six and a half minute is too much as calculation time for a move. So we have to decrease the board size. A smaller board has two advantages: First, the branching factor is reduced. Second, the simulations get shorter, thus more simulations can be performed in the same amount of time.

On a 12×12 board, the second level of the tree does contain less than 10,000 nodes instead of 204,304 (the branching factor cannot be higher than $10 \times 10 = 100$ at any step, so the average is even less). Less than 5% of the nodes remain on this level. The greedy player can perform 5,990 simulations per second, reaching 10,000 simulations in less than two seconds.

For most of the experiments (differences are marked), the 12×12 board was chosen along with 10 seconds of thinking time per move and the comparisons between two AIs are done with 200 test games, each player being White in half of them. As a game usually consists of about 15 moves, each complete test run lasts for $10 \times 15 \times 200 = 30,000$ seconds, about 8.3 hours.

5.1.3 Local Optima

Several features in the MCTS framework require the choice of a weight. So, we do not only have to evaluate the feature itself, but we also have to find an optimal weight for the feature. This is also done using test games and poses no problem. But as we get several features with different weights each, we encounter a problem. When we optimized parameter α for an AI that utilizes only feature A, we have to retune α when we also utilize feature B. But when B has to be weighted with the parameter β, we would have to find, not the best single values for α and β, but the best combination of the two.

As there can be many parameters (especially for the knowledge), it is not feasible to retune all parameters over and over again (due to the high amount of time needed for each test run). Thus, we may arrive in some local optima or even stop at a suboptimal solution (maybe the differences are too low to be recognized). But the aim of this project is not the implementation of the best performing algorithm. Rather, we want to examine different features to see which improve the strength of a TwixT AI and which do not. Therefore, we will spend some time in tuning the parameters, but they are not the main focus and we will end up with a suboptimal configuration anyway.

5.1.4 Technical Environment

For the test games, a notebook with an Intel Pentium 4 CPU 2.4Ghz is used. It has 512 MB RAM and Windows XP (Home Edition) is installed as operating system.

5.2 Comparison of Enhancements

This section shows detailed results of the experiments that have been performed. Initially, a greedy player was implemented that distributes the calculation time uniformly over all possible moves and then chooses the move with the highest score. This algorithm wins every game against a random player. This is used as a basis for the experiments and the first implementation of the MCTS algorithm is tested against it. Afterwards, each enhancement is tested by playing games between an AI with this feature and another one without it. Similarly, different parameter setups are compared. The enhancements were implemented in the order as they appear in this section.

5.2.1 Basic MCTS Player

Before we can compare the MCTS player to the greedy player and to other implementations later, we need to choose its parameter set. This includes two parameters: the MCTS threshold t (which determines when to select according to the playout strategy and when to use the MCTS selection) and the parameter c that handles the trade-off between exploration and exploitation (Formula 3.1).

The MCTS threshold is not really important: As soon as it is reached, the values of the children are taken into account. As long as there possible moves without a corresponding node, these moves will be chosen anyway. So in any case, all children will be chosen at least once. Some tests have been run, but all results were near to 50 : 50, so there are no significant differences. Thus, we simply set the threshold t to 50.

Finding a good trade-off between exploration and exploitation is much more important, as it affects the behavior of the whole search. It is hard to find a good value without running test games, as the optimal value strongly depends on several game characteristics as well as on the implementation. Further, the optimal value depends on the enhancements that have already been implemented. A value which is optimal for a MCTS player without any enhancements might not be a good choice for a player that includes several enhancements. So it is not worth to search for the optimal c value of the basic MCTS right now, because this value will not be worth a lot for the final program. So we decided to choose a reasonable value by running some simple tests. This value is $c = 2.5$. We will work on it later to find the optimal value for the final product.

For these parameters, the performance against the greedy player is given in Table 5.1. The MCTS player won 55.5% of 400 test games against the greedy player. This is not a large difference, but still significant. In addition, we can be sure that the MCTS algorithm works in the implemented way and does not weaken the player. This becomes more obvious as we take into account the extra calculation that the MCTS algorithm requires in selection, expansion and backpropagation. While the greedy player performs 5,700 stimulations per second in this environment, the MCTS player performs only 3,900 simulations per second, some 25% less. So, with fewer simulations it plays significantly better than the greedy player. This clearly indicates a working algorithm.

MCTS Player vs. Greedy Player	55.5% : 44.5%

Table 5.1: The performance of the MCTS player versus the greedy player

5.2.2 Progressive History

First, we have to decide for one of the two PH variants (compare Formulas 3.2 and 3.3). The result of an experiment with $w = 40$ (see below for an explanation of w) is shown in the Table 5.2. After 400 games, the two different PH performed similar, without a significant difference.

PH according to Formula 3.2 vs. PH according to Formula 3.3	50.0% : 50.0%

Table 5.2: The performance of the different PH variants

$w = 40$ was not chosen arbitrarily, but was the optimal value for the first PH variant (tuned beforehand). So the first variant used its optimal value in the simulation, while the second variant used the same value which may not be optimal for it. They played with an equal strength, so the second variant is better than the first and we can concentrate on it in order to find the best weight.

Using PH, we get a new parameter. Formula 3.2 (and 3.3 as well) shows that the complete PH values are weighted with the parameter w. A higher w makes the selection more dependent on the PH value, a lower w on the usual value of the node. Table 5.3 shows the experimental results for different values of w.

$w = 20$ vs. $w = 40$	58.5% : 41.5%
$w = 10$ vs. $w = 20$	52.5% : 47.5%
$w = 15$ vs. $w = 20$	60.5% : 39.5%

Table 5.3: The results for different values of w

The first experiment shows that smaller values than $w = 40$ seem to be a better choice, as $w = 20$ wins 58.5% of the games.. The next result indicates that it is not beneficial to decrease w too much, because $w = 10$ performed slightly (but not significantly) weaker than $w = 20$. And the last experiment indicates that $w = 15$ might be close to the optimum, as it results in a stronger player than $w = 20$ and

$w = 10$ do, winning 60.5% of the games against the player with $w = 20$. So $w = 15$ is the chosen value for this feature.

The result of the final experiment of the PH player against the basic MCTS player is given in Table 5.4. The PH player won 54.0% of the 400 test games, so it is significantly stronger than the basic MCTS player. So PH does not seem to give a strong improvement, but it is still a good feature to use. So we included it in the final program with $w = 15$.

PH with $w = 15$ vs. Base	54.0% : 46.0%

Table 5.4: The performance of the PH player against the basic MCTS player

5.2.3 MCTS-Solver

The only parameter needed for the implementation of MCTS-Solver is the threshold t_{Solver}. It determines when the algorithm starts checking for terminal positions. This parameter has not been tuned, but was simply set to the minimum number of moves that are required for a terminal position. For every simulation that starts with this or a higher number of moves, the terminal moves are checked.

Table 5.5 shows the result of an experiment with two PH players, one of them additionally using MCTS-Solver. We ran 1,000 games, and MCTS-Solver won 55.0% of them. So the difference that we see is significant.

MCTS-Solver vs. Base	55.0% : 45.0%

Table 5.5: The performance of MCTS-Solver playing against a basic player, both using PH

5.2.4 Reusing the Old Tree

Reusing the old tree is one of the few possible enhancements that do not require the inclusion of a weight. So we can simply run a test against the former algorithm. In the test run, both players are using PH and MCTS-Solver. Table 5.6 shows the result. The player that reused the old tree won 54.0% of 400 test games, so we have a small, but significant improvement.

Reusing Tree vs. Base	54.0% : 46.0%

Table 5.6: The performance when reusing the old tree versus not doing it, both players using PH and MCTS-Solver

5.2.5 Using Knowledge

The inclusion of knowledge is the step that requires the highest amount of testing and tuning. Most of the features included in the playout introduce at least one new parameter that we have to tune, while most parameters require at least three or four test runs to be tuned.

The first idea that is to be tested is the exclusion of the home lines during the simulations. This idea includes two steps: First, we decrease the probability of the home lines to be chosen. Second, we exclude them completely and stop the game as soon as the lines next to the home lines are connected by one of the players.

For the first step, we have to tune the parameter $w_{Exclude}$ for Formula 4.3. Different setups have been tested, all with 200 test games against the player with all above describe features, but without any knowledge. The results are shown in Table 5.7. For $w_{Exclude} \leq 30$, we have a similar behavior as without the feature. Higher values seem to increase the playing strength, but as soon as $w_{Exclude}$ exceeds 50, the strength drops. But with $w_{Exclude} = 40$, the player wins 57.5% of the games against a playing without this feature, so we have a significant improvement.

$w_{Exlude} = 20$ vs. Base	52.5% : 47.5%
$w_{Exlude} = 30$ vs. Base	44.0% : 56.0%
$w_{Exlude} = 40$ vs. Base	57.0% : 43.0%
$w_{Exlude} = 50$ vs. Base	47.5% : 52.5%
$w_{Exlude} = 60$ vs. Base	41.0% : 59.0%

Table 5.7: Different values of $w_{Exclude}$ tested against a player without knowledge

The second step does not include any kind of parameters. We simply exclude the home lines from all simulations and consider the game as over as soon as one of the players connect the lines next to his home lines. We performed only one experiment, against the player with $w_{Exclude} = 40$, and the results can be found in Table 5.8. As the table shows, the complete exclusion of the home lines further increases the player's strength, defeating the former version with 59.0%. So it is included in the further tests as well as in the final program.

Exclude All vs. $w_{Exlude} = 40$	59.0% : 41.0%

Table 5.8: The performance of a player that completely excludes the home lines against one with $w_{Exclude} = 40$

The next piece of knowledge that we implement is to increase the probability of moves that may be connected to past moves. When playing a move in a simulation, we multiply the probability of all neighbors of that move with a certain weight w_{Bridge}, valid until the end of the simulation. If we multiply with 1, the probabilities do not change. So the case $w_{Bridge} = 1$ is identical to the version without this feature. We tested different values for w_{Bridge}, with the results as shown in Table 5.9.

$w_{Bridge} = 2$ vs. Base	59.5% : 40.5%
$w_{Bridge} = 3$ vs. $w_{Bridge} = 2$	50.5% : 49.5%
$w_{Bridge} = 5$ vs. $w_{Bridge} = 2$	38.5% : 61.5%

Table 5.9: Different value for w_{Bridge} tested against one another

The first row shows that implementing the features (with $w_{Bridge} = 2$) has an impact on the playing strength as it increases the player's score to about 60%. The next row shows that values between 2 and 3 seem to result in a similar behavior. The last row shows that a value of 5 (and likely other values above) do not increase the playing strength any more, but make it weaker. So we decided to use $w_{Bridge} = 2$.

The next idea that is to be tested is the usage of different weights for "good" and "bad" bridges (see Section 4.2.1 for definition). We tried a few different values for w_{Good} and w_{Bad} for the good and bad bridges, respectively. As Table 5.10 shows, there are no significant differences. So this idea does not seem to improve the playing strength and we decided to discard it.

Now we would like to tackle the problem that there are too many performed moves in a rather small area. Therefore, we use different weights for the possible bridges: When a move is added to the probability table, we count the number of neighbors that it already has. Depending on this number, we compute the weight of the possible bridges. So w_{Bridge} is not a scalar anymore, but now it is a vector where $w_{Bridge}[n]$ is the weight for a peg with a bridge to a peg with n neighbors. The length of the vector is 8, as there are between 0 and 7 bridges possible (8 bridges are possible as well, but for $n = 8$ we do not need a weight, as there are not bridges possible anymore). Table 5.11 shows the experimental results.

Unfortunately, different weights do not change the behavior significantly. In the first experiment, we use the basic weight, but only for pegs with less than 4 neighbors. In the latter two experiments, we used higher weights for pegs with fewer neighbors. As none of these setups increases the playing strength, we decided not to use this feature.

Another idea is to vary the possible bridges' weights during the simulation. To find the optimal setup, we have to choose one of the Formulas 4.1 and 4.2, and we have to find a proper weight for it. Several test runs have been performed. Each time, one parameter setup plays against the basic version with

$w_{Good} = 3, w_{Bad} = 1.5$ vs. $w_{Bridge} = 2$	51.0% : 49.0%
$w_{Good} = 2.5, w_{Bad} = 1.5$ vs. $w_{Bridge} = 2$	50.5% : 49.5%

Table 5.10: Different values for "good" and "bad" values tested against the basic weighting

$w_{Bridge} = (2, 2, 2, 2, 0, 0, 0, 0)$ vs. $w_{Bridge} = 2$	50.5% : 49.5%
$w_{Bridge} = (3.5, 2.5, 1.5, 0.5, 0, 0, 0, 0)$ vs. $w_{Bridge} = 2$	52.0% : 48.0%
$w_{Bridge} = (2.5, 1.5, 0.5, 0, 0, 0, 0, 0)$ vs. $w_{Bridge} = 2$	51.0% : 49.0%

Table 5.11: Different value vectors for the bridges tested against the basic weighting

$w_{Bridge} = 2$. A negative w_{Count} does not mean a negative value, but means that Formula 4.2 is used instead of 4.1 for positive values.

$w_{Count} = 12$ vs. $w_{Bridge} = 2$	34.0% : 66.0%
$w_{Count} = 8$ vs. $w_{Bridge} = 2$	44.5% : 55.5%
$w_{Count} = 4$ vs. $w_{Bridge} = 2$	50.5% : 49.5%
$w_{Count} = 2$ vs. $w_{Bridge} = 2$	62.5% : 37.5%
$w_{Count} = -8$ vs. $w_{Bridge} = 2$	7.5% : 92.5%
$w_{Count} = -2$ vs. $w_{Bridge} = 2$	29.5% : 70.5%

Table 5.12: Different weights for the impact of c_{curr} tested against the basic weighting

Table 5.12 shows the interesting results of this experiment. A too large value such as $w_{Count} \geq 8$ leads to a weak performance around 40%. But choosing a better value such as $w_{Count} = 2$ enables us gain as much as 12.5%. Decreasing the weight during the simulations seems to have poor results, regardless of the weight. The first version of the formula (with increasing weights for possible bridges) is included in the final program with $w_{Count} = 2$.

As we do not want to keep playing repeatedly in the same area during the simulation, we prefer those moves that place a peg in a line where there are few other pegs yet. According to the behavior described in Section 4.2.2, we do not have a weight here. But still, there are two different versions of this idea: First, we apply the concept only to the lines orthogonal to the player's direction. As White plays from left to right, we only apply the concept on columns, and on rows for Black. This might be reasonable as it is no problem if White plays many moves in the same row as this can be a good tactic. It is only a bad idea for White to play many moves in the same column, as this will not bring him nearer to his aim. Second, we apply the concept to both kinds of lines. As Table 5.13 shows, the results of the experiment are close to 50% and thus not significant. So none of these two ideas is included in the final program or in further tests.

5.2.6 Early Game Termination

The idea of earlier game termination is, not only terminating a game when it is over, but also when a certain fraction of the board is full. Therefore, we introduce a new parameter d with which the original threshold (above which the game is terminated and considered a draw) will be multiplied. $d = 1$ does not change the behavior in comparison to the algorithm without this feature. $d > 1$ is not a reasonable setup, because then even a full board would not be considered to be a draw. If we choose a too small value for d, we may end up with a poor performance as too many games are terminated and considered draws though they would not have ended in a draw.

Different values for d have been tested, all of them against a player with $d = 1$. Table 5.14 shows the detailed results. The closer d is to 1, the closer the results are to 50 : 50. As soon as d is smaller than 0.9, the player is significantly weaker than the player that does not use this feature. So we cannot gain any strength from this feature and decided to discard it, so games are performed until the board is full.

This was the last piece of knowledge that had to be tested. Now we have a reasonable playout strategy and can use it for the last experiments. As a final experiment regarding the knowledge, we want to know

Only Orthogonal vs. Base	51.0% : 49.0%
All Kind vs. Base	51.5% : 48.5%

Table 5.13: The performance of the two different ideas that prefer rarely played lines (against a player with all described pieces of knowledge, but without this one)

$d = 0.5$ vs. $d = 1$	21.0% : 79.0%
$d = 0.7$ vs. $d = 1$	40.0% : 60.0%
$d = 0.8$ vs. $d = 1$	40.0% : 60.0%
$d = 0.9$ vs. $d = 1$	46.0% : 54.0%
$d = 0.95$ vs. $d = 1$	49.5% : 50.5%

Table 5.14: Different values for d tested, causing the simulations to be terminated and counted as draw earlier

how much the application of knowledge increases the playing strength in comparison to a player without any knowledge. Table 5.15 shows the result of this experiment. Both players used the described domain-independent enhancements (PH, MCTS-Solver etc.). In addition, one player used all the knowledge that is described above. It includes the following features and parameters: Complete exclusion of the home lines in the playouts, and increased weights for the bridges that vary during the simulation (according to Formula 4.1 with $w_{Count} = 2$). We see that the stated set of knowledge features significantly increase the playing strength. In addition, the player with knowledge performs only 1,700 simulations per second, while the enhanced MCTS player without any knowledge reaches about 2,800 simulations per second. But despite 50% more simulations, the player without knowledge is weaker.

Player using knowledge vs. Player without any knowledge	66.0% : 34%

Table 5.15: The performance of the tuned player with knowledge against the basic player without any knowledge

5.2.7 Weighting Simulations by Length

Two different approaches to modify the backpropagation were to be tested. First, we modify the simulation results before backpropagating them (according to Formula 4.4). Shorter simulations are trusted more than longer simulations. We have to choose a proper weight w_{Length} in order to get the optimal performance. Table 5.16 shows the performances of players with different parameters against the basic player (after implementing the features described in the previous sections).

$w_{Length} \geq 0.5$ seems to put a too high emphasis on the simulation length. Smaller values seem to perform slightly better than the player without this feature, but no parameter setup gives a significant difference. So this idea does not seem to improve the playing strength, and thus, was discarded.

5.2.8 Tuning c

As we have a full MCTS player including several enhancements, it is now time to tune the constant c of the UCT formula (Formula 3.1). As mentioned in previous sections, we simple used $c = 2.5$ so far. Table 5.17 lists the results of 200-game matches between players using different c values. We tested c values between -2.0 and 2.5.

For $0 < c \leq 2.5$, we did not find any significant difference. But with $c = 0$, we get a significantly stronger player. So, multiplying the exploration term in the UCT formula with zero (thereby ignoring it) gives a better performance than any value between 0 and 2. This is a quite interesting result, as the exploration feature makes sure that no good moves are “forgotten” due to some weak simulations in the beginning (which strongly depend on random values). Excluding this feature seems to be a bad idea, but still it performs better. Lee *et al.* (2009) found a similar behavior in the Go program MoGo. They

$w_{Length} = 1.0$ vs. Base	45.5% : 54.5%
$w_{Length} = 0.5$ vs. Base	47.5% : 52.5%
$w_{Length} = 0.3$ vs. Base	52.0% : 48.0%
$w_{Length} = 0.2$ vs. Base	51.0% : 49.0%
$w_{Length} = 0.1$ vs. Base	51.0% : 49.0%

Table 5.16: The performance of players with weighting by simulation length against a player without this feature

$c = 2.5$ vs. $c = 1.0$	51.0% : 49.0%
$c = 0.8$ vs. $c = 1.0$	48.0% : 52.0%
$c = 0.4$ vs. $c = 1.0$	51.5% : 48.5%
$c = 0.2$ vs. $c = 1.0$	52.5% : 47.5%
$c = 0$ vs. $c = 1.0$	56.0% : 44.0%
$c = -0.25$ vs. $c = 1.0$	57.5% : 42.5%
$c = -0.50$ vs. $c = 1.0$	59.0% : 41.0%
$c = -0.75$ vs. $c = 1.0$	73.0% : 27.0%
$c = -1$ vs. $c = 1.0$	60.0% : 40.0%
$c = -2$ vs. $c = 1.0$	36.5% : 63.5%

Table 5.17: The performance of players with different c values

even claim that $c > 0$ is useful only for preliminary implementations without enhancements, such as RAVE (we implemented PH, a variant of RAVE), but it is useless in any optimized program. According to their research, $c > 0$ becomes less useful as parameters are tuned, enhancements implemented and knowledge is included, leaving MCTS as a greedy best-first search. Using RAVE values makes sure that no strong moves are overlooked during the search. So, a better performance with $c = 0$ than with $c > 0$ is surprising, but not unreasonable.

We tried to push this though a step further. When RAVE or PH values can be used instead of the UCT exploration term, it may be even better to use a negative c constant. When RAVE / PH works well, we can even prefer the moves that have been played often, knowing that they are strong moves. In addition, the high branching factor of TwixT may lead to a situation where it is better to investigate a rather small part of the search space quite intensively than to distribute the computation time among more parts of the search space. As the results show, a negative value of c is quite a good idea. The last four rows of Table 5.17 show results with $c < 0$. As we can see, all of the tested values below zero seem to result in a better performance than $c = 1$ and even $c = 0$. But as $c = -0.75$ performed better than $c = -1$, we should not choose a too large negative value for c. For $c = -2$, the program does not play well anymore and perform even worse than for $c = 1$. We decided to use the best performing value, namely $c = -0.75$ for the following experiments, winning 73.0% against a player with $c = 1$.

Despite the given explanation, a negative c still contradicts our intuition. So we tried another idea in order to optimize the performance: Searching a small part of the search space quite intensively may be a good idea in deeper levels of the tree, but this does not make sense for the higher levels. So we defined to different c values. The first c is kept at -0.75 as it was tuned in the previous experiments. This c is labeled c_1. A different c is used for the first iteration of the selection step. For the root node, we use a constant c_2. For this constant, we tried the values 0 and 1. Table 5.18 shows the detailed results.

$c_1 = -0.75$, $c_2 = 1.0$ vs. $c_1 = c_2 = -0.75$	31.5% : 68.5%
$c_1 = -0.75$, $c_2 = 1.0$ vs. $c_1 = c_2 = 1.0$	53.0% : 47.0%
$c_1 = -0.75$, $c_2 = 0.0$ vs. $c_1 = c_2 = -0.75$	45.5% : 54.5%
$c_1 = -0.75$, $c_2 = 0.0$ vs. $c_1 = c_2 = 0.0$	53.0% : 47.0%

Table 5.18: The performance of players with different c_1 and c_2 values

As the first two rows show, the setting $c_1 = -0.75$, $c_2 = 1.0$ leads to a similar performance like the basic setting with $c_1 = c_2 = 1.0$. So, choosing a smaller value of c in nodes different from the root node does not increase the playing strength significantly. The last two rows show that the setting $c_1 = -0.75$, $c_2 = 0.0$ seems to result in a playing strength somewhere in between the strengths of $c_1 = c_2 = 0.0$ and $c_1 = c_2 = -0.75$. It is likely that choosing a smaller, even negative c in all nodes except the root node increases the playing strength slightly, but that the root node should have rather small c value as well to get the optimal performance. As we do not get a better performance with different c values at different search levels, we simply use $c = -0.75$ in further tests as well as in the final program.

5.2.9 Modifying Backpropagation by Discount Rate

The second approach in modifying the backpropagation is to use a discount rate to make older simulations less important than newer simulations. Before a new simulation result is added to the node's score, the old score is multiplied by the discount rate $rate_d$. The case $rate_d = 1$ is equal to the usual backpropagation. The experiments with a discount rate in the backpropagation were done both for $c = 1.0$ (a reasonable value for c, in order to keep the idea generic) and for $c = -0.75$ (the value with the strongest performance). In order to find the best value for $rate_d$, we tested several parameters. For all of them a test run was performed against the basic version with $rate_d = 1$. The results of 200-game matches for $c = 1.0$ and $c = -0.75$ can be found in Tables 5.19 and 5.20, respectively.

$rate_d = 0.9925$ vs. Base	55.5% : 44.5%
$rate_d = 0.99$ vs. Base	53.0% : 47.0%
$rate_d = 0.9875$ vs. Base	59.5% : 40.5%
$rate_d = 0.9825$ vs. Base	61.5% : 38.5%
$rate_d = 0.98$ vs. Base	55.5% : 44.5%
$rate_d = 0.975$ vs. Base	59.5% : 40.5%

Table 5.19: The performance of players using a discount rate, both using $c = 1.0$

$rate_d = 0.99$ vs. Base	48.0% : 52.0%
$rate_d = 0.9875$ vs. Base	53.0% : 47.0%
$rate_d = 0.985$ vs. Base	51.5% : 48.5%
$rate_d = 0.9825$ vs. Base	44.5% : 55.5%
$rate_d = 0.98$ vs. Base	45.5% : 54.5%
$rate_d = 0.975$ vs. Base	54.5% : 45.5%
$rate_d = 0.95$ vs. Base	59.0% : 41.0%

Table 5.20: The performance of players using a discount rate, both using $c = -0.75$

The experiments with both players using $c = 1.0$ produced quite interesting results. If $rate_d$ is close to 1, we get a behavior similar to the playing without a discount rate (or $rate_d = 1$). As we decrease $rate_d$, the feature has a higher impact and increases the playing strength. The best performance is achieved with $rate_d = 0.9825$, winning 61.5% of the test games against the player without a discount rate. With this setting, we get a significantly increased playing strength. When we decrease $rate_d$ too much, we do not get a strong performance anymore, as we discard too much information. As we want to optimize the playing strength for $c = -0.75$, we do not search for the optimal value of $rate_d$ very thoroughly. Maybe a value below 0.975 may be even better. But we know at least that, with $c = 1$, we can increase the strength by using a discount rate.

With $c = -0.75$, we get different results. For some of the tested $rate_d$ values, we get a result slightly above 50%, but this is not significant. When $rate_d$ is too small, it even weakens the player significantly (with $rate_d = 0.95$, it wins only 41.0%). So with a unusual, negative c value, we cannot gain anything with a discount rate. This might be due to the fact that a negative c and a discount rate both affect the selection process. So implementing one of these features may make the other obsolete. In the final program, we decided to use $c = -0.75$ and $rate_d = 1$, because the impact of the discount rate was lower than the impact of the c value.

5.2.10 Final Program

With the described features and enhancements (including the tuned parameters), we have a final program. This artificial TwixT player is called Twixter. It is the program that is used in the experiments in the following sections. This section states the precise setup of Twixter.

Twixter is based on a MCTS player using the UCT Formula 3.1. The threshold t was set to 50, and the constant c was decided to be -0.75. It uses PH according to Formula 3.3 with $w = 15$. It further uses the concept of MCTS-Solver and reuses the MCTS tree of the last move's search. In addition, it includes the pieces of knowledge that are described in the last paragraph of Section 5.2.6. It does not utilize a discount rate in the backpropagation.

After the enhancements of the algorithm have been tested and the parameters have been tuned, we get the final program. This program is able to run 2,100 simulations per second. It is interesting to play against it in order to estimate its playing strength against a human amateur. Further, it might be visible what the strengths of the program are and what kind of problems there might be.

The author of this thesis is able to defeat Twixter. Some positions require some sincere thought, but usually it is still possible to see a good move quite fast. But as soon as the author starts making one or two mistakes, the program uses these mistakes and wins the game. So the strength of the program is on the level of a human beginner.

If Twixter begins, it almost always chooses a move near the center of the board, which is actually a good choice. But when it comes to responding to enemy moves, the program has some difficulties due to the fact that no complex knowledge has been built in. So it happens that it responds to an opponent move with a rather weak move.

5.3 Playing Against Other Programs

There are several TwixT playing programs available. Against two of them Twixter was tested, namely T1j and the Moesker program. Unfortunately, there is no interface between these programs and ours, so the games have to be played manually. This reduces the number of games that we can play, but still gives us some insight into the strength of the programs.

5.3.1 T1j

T1j is a program written by Schwagereit (2010). It uses a pattern-based approach instead of a search-based one, so it is interesting to see them playing against each other. Due to a lack of automated interfaces, we had to perform the test games against T1j manually. As the time was limited, we decided to play four games – each player played two times as White and two times as Black. We played on the 12×12 board and each player had 10 seconds to find the strongest move.

T1j won three of four games against Twixter. Twixter won one of the games where it was playing White. When watching the games, it was obvious that T1j played stronger, especially when it came to positions where patterns could be applied. But it was also obvious that the games were not trivial for T1j, which won one of the games. So in comparison with T1j, Twixter is a serious opponent, but is still weaker.

5.3.2 Program by Moesker

The program of Moesker (2009) is also search based, but is also different from the program that we built. It is based on the alpha-beta algorithm with different enhancements (See referenced work for details). Again, we could only let the two programs play against each other manually. As against T1j, four test games have been performed. Although the program by Moesker is able to play on different board sizes, we had only a compiled program that was able to play on the 8×8 board. Further, the program by Moesker does not use a constant amount of time for every move, but the time per move depends on a maximum amount of time for the entire game. On each move, a certain part (one fourth) of the remaining time is used. As this behavior is not possible in Twixter, we decided to set the time for our program to 60 seconds per move. On average, this is about the same amount that the program by Moesker uses. Still, Twixter has a disadvantage here, because the program by Moesker has more time in the first moves, which are the most important one. But that is not a problem, as we are only interested in an overview of the playing strength. So we also ignored the fact that all parameters of TWIXTER were tuned for the 12×12 board with 10 seconds per move. The only parameters that was modified is the c constant in the UCT formula: Instead of $c = -0.75$, we used $c = 0$. This resulted in a quite greedy player, but it was enough to get decisive results.

All four games have been won by TWIXTER. In the long run, the program by Moesker might have won a few games, but we do not believe that there will be many. Some of its moves looked sound, but often it missed an obvious opportunity to decide the game. So even with the explained disadvantages, TWIXTER defeats the program by Moesker.

5.4 Chapter Conclusions

As already explained in Sections 1.2.2 and 4.1.2, it is not easy to build a strong TwixT player. The basic MCTS search does rarely more than a 2-ply search (See Section 5.1.2). Still, it is significantly stronger than a greedy player that utilizes simulations as well. Domain-independent enhancements increase the player's strength in the domain of TwixT, but the difference is rather small. RAVE/PH, MCTS-Solver and the reusing of the old tree all increase the playing strength significantly, but all of them reach about 55% wins in test games against a player without the feature.

The knowledge-based enhancements of selection and playout seem to work better. The implemented and tested feature mainly concentrate on the exclusion of unattractive areas of the board and on a well-weighted preferring of moves that lead to new bridges. All these features together give a total performance of 66 % : 34% in comparison to the best setup without any knowledge.

An interesting result is the effect of the modified backpropagation. Weighting simulations by their length, thus putting more trust in shorter simulations, did not improve the playing strength. But a quite simple modification during the update of a node increases the win percentage against a player without this modification to 60%: When a node is updated, we first multiply both visit count and value by a certain discount rate. After that, we add the new value and increase the counter. This may improve the strength of the player because it puts more trust in the latter simulations and does not take much into account the earlier simulations. This strategy is also easy to implement for other domains, as the update of the nodes is the same procedure for all domains. It will be interesting to test this feature for other games. This feature did not increase the playing strength anymore when the c constant was set to -0.75. But as there may be many games where a negative c value is not a good idea, in these games the backpropagation rate can be utilized.

Despite the different enhancements that were tested and included, the overall strength of the final program is not above that of an amateur. The author of this thesis is still able to win most of the games against the final program. The final program TWIXTER lost three of four test games against T1J, a pattern-based TwixT program by Schwagereit (2010). It won all four test games against the program by Moesker (2009), even with some disadvantages due to the time setting and the board setup.

Chapter 6

Conclusions

This chapter describes the conclusions of the entire research project. It revisits the problem statement and research questions, describes the encountered problems and their solutions and gives an outlook on future research.

Chapter contents: Introduction – Revisiting Research Questions, Revisiting Problem Statement, Future Research

6.1 Revisiting Research Questions

Research Question 1: *Which data structure can we use for TwixT in order to achieve the best performance with MCTS?*

The first four sections of Chapter 2 provided an answer to this question. One aspect of the implementation is the memory space. As it is always limited, we cannot use as much space as we want. So we should not store more data than we need, thus avoid redundancy. But the most important aspect of the implementation of the rules is the speed. So a redundant data structure may be the best choice when it allows a faster computation. In the case of TwixT, we decided to store the past moves by memorizing their coordinates in the correct order. This allows us to restore an arbitrary situation.

The most complicated part of the TwixT rules is the check for possible bridges: When a player places a peg, he may be allowed to add a bridge between this peg and another one. There are two requirements: First, the new peg must be a "neighbor" of an already existing peg (of the same color), i.e., the x- and y-coordinates must have a difference of 1 and 2, respectively (or vice versa, 2 and 1, respectively). If that is the case, a further check has to be performed. The considered bridge may already be blocked by another bridge (of the same or of another color). In the chosen implementation, we first look for neighbors of the two pegs. If none of these neighbors are found, there is no blocking bridge. If there are pegs found, we have to check their bridges and compare them to the considered bridge. If the bridges cross each other, the considered bridge is not possible. The described data structure allows to run about 3,900 simulations per second on the 12×12 board for the plain MCTS player.

Research Question 2: *Which is the fastest way to check for a terminal position?*

The second research question is interrelated with the first one: The implemented underlying data structure determines possible ways to check for a terminal position. Section 2.5 described the algorithm that is used to detect terminal positions. As the complete board situation is given by a list of coordinates (the coordinates of the set pegs in the correct order), it would have been possible to restore all information that we need from the move list after every move and implement an iterative algorithm that tries to find a path between the two home lines. But this check is rather slow. Instead, we decided to implement an additional list of IDs. All pegs with the same ID are connected, pegs with a different number are not. When a new bridge is added, all pegs with the

ID of one of the directly connected pegs get the same ID. As these IDs are not recalculated after every move, but are incrementally kept up to date, the check for terminal positions can be done in $O(n)$ where n is the number of placed pegs on the board. With the implemented check for terminal positions, we spend only 3.3% of the search time (4.8% of the time spent in the simulations) in the plain MCTS player.

Research Question 3: *Which domain-independent enhancements of MCTS are useful for TwixT?*

The domain-independent enhancements were discussed in Section 3.3 and the experimental results can be found in Chapter 5. We implemented and tested several enhancements: RAVE / PH, MCTS-Solver, Reusing the MCTS Tree, and Backpropagation with a Discount Rate. The implementation of RAVE / PH required some effort, as well as the tuning of the parameters. Finally, we got a small but significant increase in the playing strength (winning 54.0% of the test games against a player without this enhancement). Implementing MCTS-Solver was also possible, but we had to exclude most of the moves from being checked. Otherwise we would have spent too much time in the solving and to less time in the actual search. MCTS-Solver increased the strength of the program as well, but again slightly (winning 55% of the test games). Reusing the MCTS tree of the last search has not yet been tested in domains different from TwixT, but it seems to be promising. Only a small part of the tree could be reused, but this was enough to make a significant difference (winning 54.0%), similar to the results of PH and MCTS-Solver. An advantage of this enhancement is the fact that it is quite easy to be implemented. Using a discount rate in the backpropagation was also tested for the first time in TwixT. The idea and implementation is quite simple, as it requires nothing more than adding two lines of code – with a well-chosen parameter – in the function that handles the updates of the nodes. It turned out to be quite interesting, as it could increase the playing strength by about 6.0%. Another interesting result was the optimal setting of the constant c in the UCT formula. First, we found that $c = 0$ is significantly better than any $c > 0$. Then, we also tested negative values for c and were surprised to learn that a player with $c = -0.75$ wins 73.0% of the games against a player with $c = 1$ (while a player with $c = 0$ wins "only" 56.0% against a player with $c = 1$). It will be interesting to learn whether this is also a good idea in other domains. Unfortunately, the optimized c value caused the effect of the discount rate in the backpropagation to decrease.

Research Question 4: *What kind of TwixT knowledge can significantly improve the MCTS program?*

Chapter 4 focused on the inclusion of TwixT knowledge in MCTS. The knowledge is applied in the selection step as well as in the playout step of the MCTS procedure. As TwixT is a connection game, a high emphasis was put on preferring moves that may lead to new bridges. These moves are probably strong ones, but not in all cases. Assigning a positive weight to moves that may result in new bridges gave an increased playing strength (winning 59.5%). The same is true about increasing this weight during the simulations (winning 62.5%). Others tested features did not increase the playing strength, such as preferring horizontal or vertical bridges, assigning weights according to the number of already placed neighbors. Another implemented feature was preferring lines that have not been used often, but this feature did not increase the playing strength as well. Further, we tested different approaches to exclude bad moves from the simulations. These moves were the ones in each player's home lines. These are not necessary to decide the game, only to finish it in the end. With this feature, we could again raise the playing strength (winning 59.0%). Another feature that did not help to increase the playing strength is considering a game as a draw before the board is nearly full.

As already mentioned, the implemented pieces of knowledge were not exhaustive. But we found some interesting features that enabled us to increase the player's strength. The implementation of all the described pieces of knowledge enables a player to win 66.0% of the games against a player that is using a plain random simulation strategy.

6.2 Revisiting Problem Statement

Problem Statement: *How can we implement a Monte-Carlo Tree Search TwixT player that defeats strong human amateur players?*

By answering the research questions, we can also answer the initial problem statement. A strong AI TwixT player can only be build using domain-independent features combined with domain knowledge. Using PH and MCTS-Solver and reusing the search tree were combined with different piece of domain knowledge, mostly regarding the concept of bridges. The setup described Chapter 5 is most efficient setup of our TwixT-playing program TWIXTER.

Unfortunately, our program TWIXTER is not able to defeat strong human amateurs. On the rather small 12×12 board that we used, it is still possible for a human amateur such as the author of this thesis to defeat the program in most of the games. Still, we built a reasonable strong MCTS TwixT program that may serve as a basis for future research, where some other features may be included in order to further improve the strength of TWIXTER.

6.3 Future Research

Besides the questions answered in this thesis, there are many questions more, both in computer TwixT and in Game AI generally. This section states some of the open questions and how they could be answered.

6.3.1 Open Questions Regarding TwixT

We contributed some pieces of information to the research on the domain of AI in TwixT. We were able to build a reasonable playing TwixT program by implementing several domain-independent enhancements to MCTS as well as different feature based on domain knowledge. The experimental results can be used as a basis for future research on TwixT. Moreover, there are some parts of the program that can be worked on to gain more knowledge about AI in TwixT and to increase the playing strength of the created AI player.

Fast checking the rules enables us to run many simulations in a fixed amount of time, so we put a high emphasis on the runtime efficiency of our program. There may be several possibilities to decrease the amount of time needed to evaluate the TwixT rules. This may especially be true for the check of blocking bridges when a new bridge is to be added. This part of the rules is quite complicated from a programmer's angle. So it would be interesting to find a faster approach.

Another area where might be some research useful is the inclusion of more knowledge in the playout (and in the selection). We concluded from our results that building bridges is an important concept of TwixT and this should be addressed in the playout strategy. Some ideas how to address this concept were proposed and tested, giving a clue where to search for other ideas. But also outside the concept of building bridges it is interesting to know how the playout strategy can further be improved. Some of these ideas were also tested, but only few of them with positive results.

6.3.2 Open Questions Regarding General Game AI

As we also included several enhancements that are domain-independent, we contributed to the general research on AI in classic board games. We could affirm the usefulness of enhancements like RAVE / PH and MCTS-Solver in the domain of TwixT. Further, we could show that tuning parameters is an important part of the implementation of an AI.

There were also some enhancements newly introduced by us. One of these enhancements is reusing the search tree from the previous search process. This idea is quite easy to implement, as it has only to be check if the current board state is somewhere in the tree of the former search. If so, we do not use an empty tree, but use the corresponding subtree. The effect of this enhancement strongly depends on the branching factor of the domain and on the current search quality. A high branching factor makes this enhancement less effective, as only a smaller part of the old tree can be used again. A higher search quality makes this enhancement more effective, as then a higher percentage of the old search tree will be in the subtree that is reused. We implemented this enhancement quite early in the research, when the search was not good yet. In addition, TwixT has a high branching factor, making this enhancement even less effective. But despite this two factors, we still got a significant increase in the playing strength. That makes us conclude that this idea will work even better in other domains, especially when the search is already quite sophisticated.

Another idea that was not proposed before in the inclusion of a discount rate in the node update. When updating a node during the backpropagation – before adding the backpropagated value – we multiply the counter and the current score (the sum of all values backpropagated yet) by a discount rate slightly below 1. By doing that, we trust older simulations less than newer simulations. This concept may also work in other domains, as the backpropagated results get more and more realistic during the search and the newer simulations should be trusted more – independent from the domain. So it is an interesting question if this enhancement will also work in other domains and if so, how much we can gain through it. This idea did not work with the negative c constant, but for those games where the optimal c is non-negative, the discount rate might increase the playing strength.

Another thing was especially interesting to learn and may be generalized to other domains. When PH values are used, it is not necessary to include the UCT exploration term anymore. It may even be useful to include it with a negative weight, preferring moves that have already been played often. This might work due to the fact that the simulations are already realistic and the PH values are already a good estimate. So there is no reason to limit the search for the optimal c to values greater than zero.

References

Cazenave, T. and Borsboom, J. (2007). Golois Wins Phantom Go Tournament. *ICGA Journal*, Vol. 30, No. 3, pp. 165–166. [4]

Cazenave, T. and Saffidine, A. (2009). Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, Vol. 23, Nos. 2–3, pp. 183–202. In French. [4, 5]

Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2007). Progressive Strategies for Monte-Carlo Tree Search. *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)* (eds. P. Wang and others), pp. 655–661, World Scientific Publishing Co. Pte. Ltd. [16, 17, 18]

Chaslot, G.M.J-B., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [4, 15, 16]

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science (LNCS)*, pp. 72–83, Springer-Verlag, Heidelberg, Germany. [4, 15, 17]

Finnsson, H. and Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008* (eds. D. Fox and C.P. Gomes), pp. 259–264, AAAI Press. [4]

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. *Proceedings of the International Conference on Machine Learning (ICML)* (ed. Z. Ghahramani), pp. 273–280, ACM. [18]

Hsu, F.-h. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. [1]

Junghanns, A. (1998). Are there Practical Alternatives to Alpha-Beta? *ICCA Journal*, Vol. 21, No. 1, pp. 14–32. [4]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [4]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Artificial Intelligence*, pp. 282–293. [4, 15, 17]

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 73–83. [35]

Lorentz, R.J. (2008). Amazons Discover Monte-Carlo. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science (LNCS)*, pp. 13–24. [4]

Moesker, K. (2009). TwixT: Theory, Analysis and Implementation. M.Sc. thesis, Maastricht University, Maastricht, The Netherlands. [4, 6, 12, 29, 38, 39]

Nijssen, J.A.M. and Winands, M.H.M. (2010). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010).* [19]

Schwagereit, J. (2010). Playing Twixt. "http://www.johannes-schwagereit.de/twixt/index.html". [1, 2, 6, 38, 39]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275. [1]

Teytaud, F. and Teytaud, O. (2010). Creating an Upper-Confidence Tree Program for Havannah. *Advances in Computer Games (ACG 2009)* (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of *Lecture Notes in Computer Science (LNCS)*, pp. 65–74, Springer-Verlag, Heidelberg, Germany. [5]

Turing, A.M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B.V. Bowden), pp. 286–297, Pitman Publishing, London, England. [1]

Winands, M.H.M. and Björnsson, Y. (2010). Evaluation Function Based Monte-Carlo LOA. *Advances in Computer Games (ACG 2009)* (eds. H.J. van den Herik and P.H.M. Spronck), Vol. 6048 of *Lecture Notes in Computer Science (LNCS)*, pp. 33–44, Springer-Verlag, Berlin Heidelberg, Germany. [4]

Winands, M.H.M., Björnsson, Y., and Saito, J-T. (2008). Monte-Carlo Tree Search Solver. *Computers and Games (CG 2008)* (eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science (LNCS)*, pp. 25–36. [19]

Appendix A

Test Games Against Other Programs

In Computer TwixT, a move consists only of the coordinate of the placed peg. All possible bridges are added automatically, no bridges are removed. White plays from top to bottom, where the x-coordinate is given by a letter and the y-coordinate by a number.

A.1 Test Games Against T1j

The test games against T1J have been played on the 12×12 board.

A.1.1 T1j as White

First game:
1. H6 2. G4 3. H11 4. K4 5. G3 6. E3 7. I4 8. H7 9. G8 10. G2 11. H1 12. F10 13. H10. 14. Gives up, White (T1J) wins.

Second game:
1. H6 2. H7 3. C7 4. E4 5. D5 6. D2 7. F4 8. D9 9. D10 10. B10 11. E8 12. G3 13. H3 14. I2 15. J2 16. F10 17. E12 18. Gives up, White (T1J) wins.

A.1.2 Twixter as White

First game:
1. H8 2. F4 3. J4 4. I6 5. I5 6. H5 7. G6 8. I3 9. I2 10. D8 11. K6 12. G11 13. K11 14. E10 15. J9 16. G9 17. I12 18. K7 19. G1, White (TWIXTER) wins.

Second game:
1. G7 2. G4 3. D4 4. C8 5. H9 6. E3 7. J4 8. I3 9. I2 10. J5 11. E2 12. D5 13. C6 14. E7 15. K6 16. K7 17. C10 18. A7 19. Gives up, Black (T1J) wins.

A.2 Test Games Against the Program by Moesker

The test games against the program by Moesker have been played on the 8×8 board.

A.2.1 Moesker as White

First game:
1. D3 2. D7 3. E1 4. E5 5. G4 6. G6 7. B6 8. C5 9. C4 10. B7 11. Gives up, Black (TWIXTER) wins.

Second game:
1. D3 2. D5 3. F6 4. F4 5. B6 6. B4 7. G4 8. H3 9. Gives up, Black (TWIXTER) wins.

A.2.2 Twixter as White

First game:
1. E4 2. F4 3. G3 4. H3 5. D6 6. F2 7. D2 8. F5 9. E8 10. Gives up, White (Twixter) wins.

Second game:
1. F5 2. E5 3. D4 4. C2 5. G1 6. G6 7. E6 8. F7 9. F8 10. E3 11. F3, White (Twixter) wins.