Monte-Carlo Tree Search for the Simultaneous Move Game Tron

N.G.P. Den Teuling

June 27, 2011

Abstract

Monte-Carlo Tree Search (MCTS) has been successfully applied to many games, particularly in Go. In this paper, we investigate the performance of MCTS in Tron. Tron is a two-player simultaneous move game. We try to increase the playing strength of an MCTS program for the game of Tron by applying several enhancements to the selection, expansion and play-out phase of MCTS.

From the experiments, we may conclude that Progressive Bias, altered expansion phase and play-out cut-off all increase the overall playing strength, but the results differ per board. MCTS-Solver appears to be a reliable replacement for MCTS in the game of Tron, and is preferred over MCTS for its ability to search the tree for a proven winning move sequence. The MCTS programs tested performed poorly against an $\alpha\beta$ program that used a sophisticated evaluation function, indicating that the MCTS programs play far from perfect and that there is a lot of room for improvement.

1 Introduction

Over the past fifty years, much work has been done in the field of games and Artificial Intelligence. Board games such as Chess, Go and Checkers have been popular research topics. The rules of these games are clear and their environments are simple. The challenge of these games lies in finding strong moves in the large state space. The common way of searching the state space is using $\alpha\beta$-search [12] with a domain-specific evaluation function. However, for games with a large state-space that require a deep search or a complex positional evaluation function, this approach might be undesirable. An alternative approach is Monte-Carlo Tree Search (MCTS) [5, 6, 13]. In contrast to $\alpha\beta$-search, MCTS does not require a positional evaluation function as it relies on stochastic simulations. MCTS has proven itself to be a viable alternative in, for instance, the board games Go [6], Havannah [8], Hex [1], Amazons [14] and Lines of Action [22].

A challenging new game is Tron. It is a two-player game that bears resemblance to Snake, except that in Tron, players leave a wall behind at each move. An interesting aspect of Tron is that it is a simultaneous move game, rather than the usual turn-based game. In this paper, the performance of MCTS in Tron is investigated, continuing the pioneering work performed by Samothrakis *et al.* [16]. We examine approaches to improve the program's playing strength, by trying out different evaluation functions and MCTS enhancements.

In 2010, the University of Waterloo Computer Science Club organized an AI tournament for the game of Tron [20]. Overall, the MCTS programs were outperformed by $\alpha\beta$ programs. It is therefore interesting to compare the playing strength of the MCTS programs with several enhancements described in this paper against the top $\alpha\beta$ program of the tournament.

The research questions of this paper are:

- How does the play-out strategy affect the playing strength of an MCTS program?
- Which search enhancements increase the playing strength of the MCTS program?
- Does an MCTS-Solver program perform better than an MCTS program in the game of Tron?
- How does the performance of the MCTS program compare to an $\alpha\beta$ program for the game of Tron?

This paper is organized as follows: Section 2 gives a brief description of the game of Tron and the difficulties that the program has to be able to handle, to play at a decent skill level. Section 3 explains the MCTS and MCTS-Solver method applied to Tron. The possible enhancements applied to MCTS regarding the selection strategy are described in Section 4, followed by an enhanced expansion strategy in 5. Play-out strategies are described in Section 6. Experiments and results are given in Section 7. Finally in Section 8, conclusions from the results are drawn and future research is suggested.

2 The Game of Tron

The game of Tron originates from the movie *Tron*, released by Walt Disney Studios in 1982. The movie is

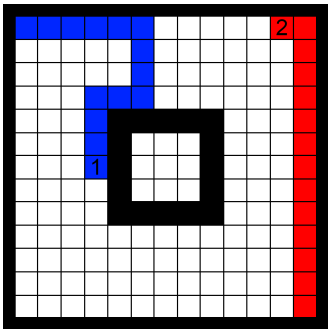

Figure 1: A Tron game on a board with obstacles after 13 moves. The blue player (1) has cut off the upper part of the board, restricting the space the red player (2) can fill.

about a virtual world where motorcycles drive at a constant speed and can only make 90 degree angles, leaving a wall behind them as they go. The game of Tron investigated in this paper is a board version of the game played in the movie.

Tron is a two-player board game played on an $m \times n$ grid. It is similar to Snake: each player leaves a wall behind them as they move. In Snake, the player's wall is of a limited length, but Tron does not have such a restriction. At each turn, the red and blue player can only move one square straight ahead, or to the left or right. Both players perform their moves at the same time, they have no knowledge of the other player's move until the next turn. Players cannot move to a square that already contains a wall. If both players move to the same square, it is considered a draw. If a player moves into a wall, he loses and the other player wins. Usually the boards are symmetric, such that none of the players has an advantage over the other player. A typical board size is 13×13.

The game is won by outlasting your opponent such that the opponent has no moves left other than moving into a wall. At the early stage of the game, it is difficult to find good moves as the number of possible move sequences is quite large and it is difficult to predict what the opponent will do. Boards can contain obstacles (see Figure 1), further increasing the difficulty of the game because filling the available space becomes a more difficult task. Obstacles can provide opportunities to cut off an opponent, reducing the opponent's free space while maximizing your own.

3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search method that constructs a search tree using many simulations (called play-outs) [5]. Play-outs are used to evaluate a certain position. Positions with a high winning percentage are preferred over those with a lower winning percentage. MCTS constructs a search tree consisting of nodes, where each node represents a position of the game. Each node i has a value v_i and a visit count n_i.

The search starts from the root node, which represents the current position. The tree is explored at random, but as the number of simulated nodes increases, it gains a better evaluation of the nodes and can focus on the most promising nodes (exploitation).

Although Tron is a simultaneous move game, it is possible to represent it as a turn-based game. The MCTS program is always first to move inside the tree, followed by the other player. An issue arises when the players can run into each other. A solution is given in Subsection 3.1.

MCTS is divided into four phases [5]. These phases are executed until the move-selection time is up. The phases are illustrated in Figure 2. We explain the phases in detail below.

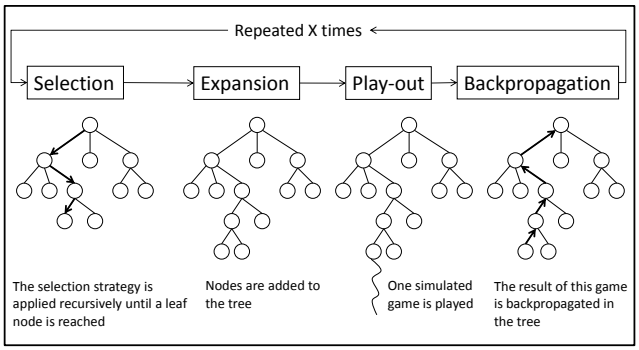

Figure 2: Outline of the Monte-Carlo Tree Search [5].

Selection In the selection phase, a child node of a given node is selected according to some strategy until a leaf node is reached. The selection task is an important one, as the goal is to find the best move. Because moves are evaluated by simulation, promising nodes should be played (exploited) more often than unpromising nodes. However, to find these nodes, unvisited nodes have to be tried out as well (exploration). Considering that in Tron, a player has at most 3 different moves at any turn (except for the first turn, where there could be 4 moves), this is not a problem. Since the number of simulations that can be performed is limited, a good balance has to be found between exploring and exploiting nodes. The simplest selection strategy is selecting a child node at random.

One selection strategy that provides a balance between exploration and exploitation, is UCT (Upper Confidence Bound applied to Trees) [13]. It is based on the UCB1 algorithm (Upper Confidence Bound) [2], which originates from the Multi-Armed Bandit

problem [15]. Given that the play-outs are reliable and a sufficient number of play-outs have been simulated, UCT will converge to the game-theoretic value of the position. UCT has been successfully applied in the game of Go [9, 10]. UCT selects a child node k from node p as follows:

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \qquad (1)$$

C is a constant, which has to be tuned experimentally. Generally, UCT is applied after the node has first been visited a certain number of times T, to ensure all nodes have been sufficiently explored to apply UCT. If a node has a visit count less than T, the random-selection strategy is applied [6].

Expansion In the expansion phase, the selected leaf node p is expanded. Since the number of child nodes is at most 3 in Tron, all nodes are added. The selection strategy is then applied to node p, returning the node from which the play-out starts.

Play-out In this phase, the game is simulated in self-play, starting from the position of the selected node. Moves are performed until the game ends, or when the final can be estimated reliably. In contrast to the selection phase, both players move simultaneously in the play-out phase. The strategy used for selecting the moves to play can either be performing random moves, or using domain-specific knowledge that increases the quality of the simulation. The play-out phase returns a value of 1, 0 or -1 for the play-out node p, depending on whether the simulated game resulted in a win, draw, or loss, respectively. The same values are awarded to end-nodes in the search tree.

If the play-out node belongs to the program, the other player will first perform a move, such that both players have performed the same number of moves.

Backpropagation The result of the simulation is back-propagated from the leaf node all the way back to the root node of the search tree. The values of the nodes are updated to match the new average of the play-outs.

After the simulations, the final move is selected. This is the move that will be played by the program. The move is selected by taking the most 'secure' child of the root node [5]. The secureness of a node i is defined as: $v_i + \frac{A}{\sqrt{n_i}}$, where A is a constant. In the experiments, based on trial-and-error for the MCTS program, $A = 1$ is used.

3.1 Handling Simultaneous Moves

Treating Tron as a turn-based game inside the tree works out quite well in almost every position of the game. However, if a position arises where the MCTS program has the advantage, but the players are at risk of crashing into each other, the program might play the move that leads to a draw. This happens because inside the search tree, the root player is always the first to move (as done in [16]). Since the root player already moved to the square that was accessible to both, the non-root player can no longer move to this square.

This problem is solved by adding an enhancement to the expansion strategy: if a node n belongs to the root player, and the non-root player could have moved to the square the root player is currently at, a terminal node is added to n with the value of a draw (i.e., 0). An example is shown in Figure 3.

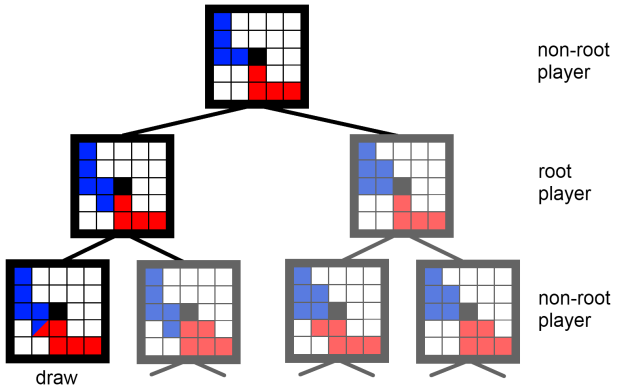

Figure 3: A game tree of Tron, illustrating how simultaneous moves are handled. In the left-most branch, both players moved to the same square, resulting in a terminal node that ends in a draw.

3.2 Monte-Carlo Tree Search Solver

Monte-Carlo Tree Search Solver (MCTS-Solver) [22] is an enhancement over MCTS that is able to prove the game-theoretic value of a position. MCTS combined with UCT requires significantly more time to reach this value, since it requires a large number of play-outs to converge to the game-theoretic value. Running MCTS-Solver requires a negligible amount of additional computation time on top of MCTS. Since proven positions do not have to be evaluated again, time can be spent on other positions that have not been proven yet.

Unfortunately, MCTS-Solver as proposed by Winands *et al.* [22] does not handle draws since draws are exceptional in Lines of Action. Since draws occur often in Tron, an enhanced MCTS-Solver is used that does handle draws.

The *Score-Bounded MCTS-Solver* [4] extends MCTS-Solver to games that have more than two game-

theoretic outcomes. It attaches an interval to each node, as done in the B* algorithm [3]. The interval is described by a pessimistic and optimistic bound. The pessimistic score represents the lowest achievable outcome for the root player, and the optimistic score represents the best achievable outcome for the root player. Given enough time and information, the pessimistic and optimistic value of a node n will converge to its true value.

An advantage of Score-Bounded MCTS is that the bounds enable pruning as in $\alpha\beta$ search, skipping unpromising branches. The initial bound of a node is set to $[-1.0, 1.0]$. Score-Bounded MCTS has been shown to solve positions considerably faster than MCTS-Solver [4].

3.3 Progressive Bias

Although UCT gives good results compared to other selection strategies that do not use knowledge of the game, there is room for improvement. The random selection strategy applied when the visit count of the node is small, can be replaced by a more promising strategy. A possible strategy would be using a play-out strategy. Since the accuracy of the selection strategy increases as the number of visits increases, it is desirable to introduce a so-called 'progressive strategy'. The progressive strategy provides a soft transition between the two strategies [5]. Two popular progressive strategies are *progressive bias* and *progressive unpruning* [5]. The progressive unpruning strategy reduces the branching factor of the tree, but since the branching factor in Tron is low, this strategy is not applied.

The *progressive bias* (PB) strategy combines heuristic knowledge with UCT to select the best node [5]. By using knowledge of the game of Tron, node selection can be guided in a more promising direction, one that might not have been found by using play-outs only. These heuristics can be computationally expensive. A trade-off has to be made between simulating more games and spending more time on computing heuristics.

When few games have been played, the heuristic knowledge has a major influence on the decision. The influence gradually decreases when the node is visited more often. The PB formula is as follows: $\frac{W \times P_{mc}}{l_i+1}$ [22]. W is a constant (set to 10 in the experiments). P_{mc} is the transitional probability of a move category mc. l_i denotes the number of losses in node i, this way, nodes that do not turn out well are not biased for too long. The formula of UCT and PB combined is:

$$k \in argmax_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \frac{W \times P_{mc}}{l_i + 1} \right) \quad (2)$$

The transitional probability of each move category is acquired from games played by expert players. Since no such games are available, the probabilities are obtained from games of MCTS players. The transitional probability P_{mc} of a move belonging to the move category mc is given by:

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \quad (3)$$

$n_{played(mc)}$ denotes the number of positions in which a move belonging to move category mc was played. $n_{available(mc)}$ is the number of positions where a move belonging to move category mc could have been played.

We distinguish six move features in Tron:

- Passive: The player follows the wall that it is currently adjacent to.
- Offensive: The player moves towards the other player, when close to each other.
- Defensive: The player moves away from the other player, when close to each other.
- Territorial: The player attempts to close off a space by moving across open space towards a wall.
- Reckless: The player moves towards a square where the other player could have moved to, risking a draw.
- Obstructive: The player moves to a square that contains paths to multiple subspaces, closing off these spaces (at least locally).

The observed move categories and their transitional probabilities are given in Subsection 7.1.

4 Heuristic Knowledge in Tron

This section describes two methods of evaluating a position. The space estimation method is also used in Section 5 and Subsection 6.2.

4.1 Space Estimation

The remaining available space of a player is a useful heuristic in Tron because the game is won by filling a larger space than the opponent. Space estimation comes to use when, for instance, the program is at a square where it has to choose between two spaces. Biasing the selection towards moving to larger spaces saves time on simulating less-promising nodes that lead towards smaller spaces. We only focus on estimating the number of moves a player can make in a space that is not reachable by the other player.

Counting the number of empty squares does not always give an accurate estimation of the number of moves a program requires to fill the available space. Spaces can contain squares that can be reached, but offer no path back to fill the rest of the space.

One way to get an estimation of the available space is by filling up the space in a single-player simulation, and

counting the number of moves [17]. The simulation uses a greedy wall-following heuristic. This heuristic works as follows: A move is selected such that the player moves to a square that lies adjacent to one or more walls (excluding the wall at its current square). If any of the moves cuts the player off from the other possible moves, the available space of each move is estimated and the move leading having the largest available space is selected. If there are multiple possible moves of equal score, a move is selected at random. This method does not always give the correct number of moves, but it gives a good lower bound on the amount of available space.

Instead of counting the number of empty squares, a tighter upper bound can be obtained by treating the board as a checker board. The difference in the number of grey and white squares gives an indication of the number of moves that can be performed [7]. The estimated number of moves M is computed by: $M = Z - |c_g - c_w|$. Z is the total available space, c_g is the number of grey tiles (including the one the player is currently standing on), and c_w is the number of white tiles. The estimated number of moves can be substantially off if the space contains large subspaces that offer no way back to other subspaces. Three example spaces are shown in Figure 4.

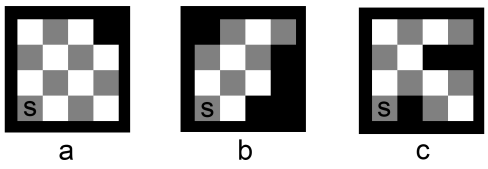

Figure 4: Three example boards where the player is isolated from the other player. The player starts at square S.

The estimated number of moves for boards a, b and c are 13, 9 and 11, respectively. The true number of moves for the boards are 13, 9 and 10. Note that the estimation of board c is off because of the two separated subspaces. Had Z been solely used as the move estimation, the estimated number of moves would have been 14, 10 and 12.

4.2 Tree of Chambers

A difficult aspect of space estimation is dealing with *articulation points*. Articulation points are squares that, once passed by a player, offer no direct way back (the player cannot turn around). Articulation points force a player to make a choice when there are more than one square to choose from. Groups of squares that are accessible via one articulation point or more are called *chambers* [11]. An example of a board containing chambers is given in Figure 5.

When evaluating a board, taking these articulation points and chambers into account will result in a more

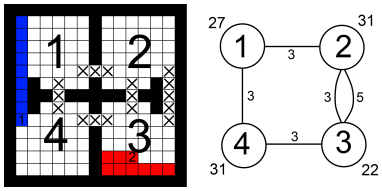

Figure 5: An example of a board and its corresponding graph. The board contains 4 chambers. Squares containing an X are articulation points. Each white region separated by articulation points is a chamber.

reliable evaluation. In general, players want to minimize the number of decisions they have to take, since these decisions usually mean that free space is left unfilled. However, there may be some sequence of articulation points and chambers that gives a larger space than a single chamber. If the players are not isolated from each other, the actions of the opponent player have to be taken into account. By generating a Voronoi diagram [21] of the board, we can see which chambers are reachable by both players and correct the expected number of available moves inside that chamber [17].

By creating a graph representation of the board, the best sequence of chambers can quickly be found since the number of chambers is generally small. In this graph, the vertices represent individual chambers and the edges their articulation points. The weight of the vertex represents the space inside the chamber, and the weight of the edge represents the length of the articulation point. The graph is built up starting at the player. The starting square counts as a chamber. If the player starts adjacent to articulation points, this starting chamber is empty.

5 Predictive Expansion Strategy

In Tron, players will often get separated from each other (if not, the game ends in a draw). If a game is in such a position, the outcome of a game can be predicted in reliable way. The *predictive expansion* strategy uses space estimation to predict the outcome of a position. (to estimate the lower and upper bound of the space, see Subsection 4.1.) It has to be noted that only nodes of the non-root player are evaluated, so both players have performed their moves when the position is evaluated.

If the outcome of a node can be predicted with certainty, the node does not have to be simulated, and is treated as a terminal node. This works as follows: the node candidate for expansion is evaluated using the space estimation heuristic. If there is a way for the players to reach each other, the node is expanded in the default way. If the players are isolated from each other and the outcome can be predicted, the node is not expanded and it is treated as a terminal node. The result of the prediction is backpropagated.

This strategy has two advantages over the old expansion strategy. Firstly, applying space estimation is faster than performing a play-out. If a sufficient number of games are cut off, the time spent on space estimation is regained, and more time can be spent on searching through other parts of the tree. Secondly, the outcome prediction is more reliable than performing multitudes of play-outs. This prevents the program from underestimating positions where one or more players are closed off in a large space.

Once the game reaches the endgame phase, the enhanced expansion strategy is no longer applied since the MCTS program is has shown to be capable of efficiently filling up the remaining space.

6 Play-out Strategies

The simplest play-out strategy is the *random-move* strategy. During the play-out, players perform a random move. Moves that result in an immediate defeat are excluded from the selection.

The author observed that play-outs performed using a random-move strategy give surprisingly good results in Tron, considering that the randomly moving Tron programs frequently trap themselves. The reliability of the play-outs can be further increased by using a more advanced play-out strategy [10].

We propose six play-out strategies. The resulting playing strength of each of these strategies is determined in the experiments.

Wall-following strategy This strategy is inspired by the wall-following heuristic described in Subsection 4.1. The strategy selects the move leading to the square with the most number of walls (but smaller than 3). If multiple of moves lead to squares of the same number of walls, one of the moves is randomly selected. A problem with the wall-following strategy is that it does not leave much room for a rich variety of simulations. During each play-out, the moves performed will roughly be the same. It means that running more simulations does not necessarily increase the accuracy of the move value.

Offensive strategy The offensive strategy selects moves that brings the player closer to the opponent player. If more than one move brings the player closer, one of the moves is selected at random. If there is no move that brings it closer to the opponent, a random move is performed.

Defensive strategy This play-out strategy selects the move that increases the distance to the opponent player. If there is no such move, a random move is performed. If more than one move increases the distance from the opponent player, one of the moves is played at random.

Mixed strategy The mixed strategy is a combination of the random play-out strategy and the previously mentioned strategies. At each move, a strategy is randomly selected according to a certain probability. The reasoning behind this strategy is that none of the strategies are particularly strong, and combining them may give better results.

The wall-following strategy has a 50% probability of being played, whereas the random-move, defensive and offensive are played 20%, 25% and 5% of the time, respectively.

Move-category strategy This strategy uses the move category statistics used by the Progressive Bias enhancement to select a move. Moves are selected by roulette-wheel selection.

ϵ-greedy strategy This strategy has a probability of $1-\epsilon$ (i.e. 90%) of playing the wall-following strategy, and a probability of ϵ (i.e. 10%) of playing a random other play-out strategy [18, 19].

6.1 Endgame Strategies

The game of Tron can be split into two phases: the phase where players try to maximize their own available free space, and the phase where the players are isolated and attempt to outlast the other player by filling the remaining space as efficiently as possible, referred to as the endgame phase.

During the endgame phase, the same strategies as mentioned above can be used, with the exception of the offensive and defensive strategy since there is no point in biasing the move on the position of the other player.

6.2 Play-out Cut-Off

Although a game of Tron is guaranteed to terminate, as each move brings the game moves closer to a terminal position, the number of moves performed during the play-out phase can be reduced. This saves time, leaving room for more simulations. Using heuristic knowledge, the result of a game can be predicted without the need to completely simulate it. A major problem with applying heuristic knowledge is that it costs a lot of computation time compared to playing moves. Therefore, the positions are only evaluated once every 5 moves. An additional advantage of predicting the play-out outcome is that the accuracy of the play-outs is increased, because the player with the largest space can still lose a portion of the simulated games due to the weak play of the play-out strategies.

The heuristic to predict the outcome of a game is the same as used in Subsection 4.1.

7 Experiments and Results

In this section, the proposed enhancements of Section 4, 5 and 6 are tested and results are given. The best-performing MCTS program of this paper is tested against the winning program of the Tron Google AI Challenge, A1K0N [17].

The experiments are conducted on a 2.4 GHz AMD Opteron CPU with 8 GB of RAM. Experiments are conducted on three different boards, each providing different difficulties for the programs. The boards are shown in Figure 6. Although all boards are symmetric, experiments are run for both colours to eliminate the possibility that the playing strength of the program is affected by its colour.

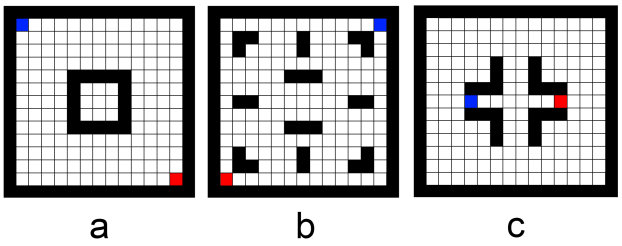

Figure 6: The three boards used in the experiments

The following settings are used for the experiments, unless mentioned otherwise: In each experiment, 100 games are played on each board for both setups. In total, 600 games are played. The time players can spend on computing the next move is 1 second. The programs have no knowledge about the move the other program will be performing.

The proposed selection strategies are tested in Subsection 7.1. Subsection 7.2 describes the results of the various play-out strategies. The Predictive Expansion strategy is reported in Subsection 7.3. The playing strength of MCTS-Solver is tested in Subsection 7.4. Finally, MCTS programs combined with the enhancements are tested against the winning program of the Tron Google AI Challenge in Subsection 7.5.

7.1 Selection Strategy Experiments

In this subsection the MCTS-UCT program is tuned and the Progressive Bias strategy is tested.

Tuning UCT

Before the other experiments are conducted, we first determine a near-optimal value for C the MCTS-UCT program. As for the minimum number of visits T before UCT is applied, $T = 30$ is used, found by trial-and-error (the value of T did not seem to matter that much). In this experiment, an MCTS-UCT program plays against a Monte-Carlo program. The Monte-Carlo program performs 1-ply search. Both programs use the random-move play-out strategy, and the secure-child method to select the final move. The Monte-Carlo program runs 110,000 play-outs per second on average, whereas MCTS-UCT only runs 85,000 play-outs per second on average.

C	2	5	8	10	20	50
Win	59%	76%	78%	86%	78%	56%

Table 1: MCTS-UCT parameter tuning

The win rates for different C values against the Monte-Carlo program are shown in Table 1. Out of all values tried for C, the best value is $C = 10$, winning 86% of the games against the Monte-Carlo program.

Progressive Bias

In this subsection, the Progressive Bias (PB) strategy is tested for different values of W. The MCTS-PB program is tested against the MCTS-UCT program. Table 2 shows the transitional probabilities, derived from self-play games by the MCTS-UCT program on the three boards (200 games per board).

Move category	P_{mc}
Defensive	28.9%
Defensive/Territorial	36.2%
Offensive	29.6%
Offensive/Reckless	0.0%
Offensive/Territorial	21.6%
Passive/Defensive	77.1%
Passive/Defensive/Obstructive	92.8%
Passive/Offensive	78.9%
Passive/Offensive/Obstructive	86.5%
Passive/Offensive/Reckless	6.6%
Passive/Offensive/Obstructive/Reckless	15.9%

Table 2: Move categories and their respective transitional probabilities, obtained from 600 MCTS-UCT games.

W	Board a	Board b	Board c	Total
0.5	45%	48%	65%	53 ± 4 %
1	49%	45%	67%	54 ± 4 %
5	55%	45%	63%	54 ± 4 %
10	48%	49%	63%	53 ± 4 %
20	36%	52%	66%	51 ± 4 %

Table 3: Win rates of MCTS-PB vs. MCTS-UCT.

Table 3 shows that Progressive Bias does not improve the playing strength on board a and b for any of the tested values of W, but it noticably increases the playing strength on board c. It is not clear why this happens. Furthermore, the value of W does not seem to matter that much.

7.2 Play-out Strategy Experiments

In this subsection, the various play-out strategies proposed in Section 6 are tested. The play-out strategies

are tested in a round-robin tournament of MCTS-UCT programs, each using a different play-out strategy. The tournament is run on all three boards. Table 4 gives the results of the individual boards. Table 5 shows the averaged results of the boards.

Board a	Rand.	Wall	Off.	Def.	Mixed	Cat.	ϵ-g.
Random		58%	90%	83%	71%	67%	52%
Wall	42%		90%	52%	21%	40%	26%
Offensive	10%	10%		31%	16%	13%	12%
Defensive	17%	48%	69%		55%	35%	32%
Mixed	29%	79%	84%	45%		49%	28%
Category	33%	60%	87%	65%	51%		38%
ϵ-greedy	48%	74%	88%	68%	72%	62%	
Board b	Rand.	Wall	Off.	Def.	Mixed	Cat.	ϵ-g.
Random		15%	100%	70%	56%	53%	58%
Wall	85%		99%	50%	78%	86%	88%
Offensive	0%	1%		0%	0%	0%	0%
Defensive	30%	50%	100%		77%	96%	51%
Mixed	44%	22%	100%	23%		37%	45%
Category	47%	14%	100%	4%	63%		36%
ϵ-greedy	42%	12%	100%	49%	55%	64%	
Board c	Random	Wall	Off.	Def.	Mixed	Cat.	ϵ-g.
Random		91%	98%	18%	43%	50%	49%
Wall	9%		81%	15%	7%	44%	9%
Offensive	2%	19%		9%	20%	12%	15%
Defensive	82%	85%	91%		76%	80%	90%
Mixed	57%	93%	80%	24%		40%	50%
Category	50%	56%	88%	20%	60%		57%
ϵ-greedy	51%	91%	85%	10%	50%	43%	

Table 4: Play-out strategy results on board a, b and c

	Random	Wall	Off.	Def.	Mixed	Cat.	ϵ-g.
Random		55%	96%	57%	56%	57%	53%
Wall	45%		90%	39%	35%	57%	41%
Offensive	4%	10%		13%	12%	8%	9%
Defensive	43%	61%	87%		69%	71%	58%
Mixed	44%	65%	88%	31%		42%	41%
Category	43%	43%	92%	29%	58%		44%
ϵ-greedy	47%	59%	91%	42%	59%	56%	

Table 5: Averaged play-out strategy results of all boards

The results show that the boards have a large influence on the effectiveness of the strategies. As such, it is difficult to select the best strategies based on these results. Overall, the random-move, defensive and wall-following strategies seem to be the best strategies. The random-move strategy performs well on all boards, whereas the defensive strategy stands out on board c. The wall-following strategy works well against the random-move strategy on board b. The random-move strategy is used in the experiments of the next subsections.

Play-out Cut-off

The play-out cut-off (PC) enhancement is tested by matching the MCTS-PC program against the MCTS-UCT program. MCTS-PC runs considerably less play-outs (25,000 per second on average) due to the computation time required by the play-out cut-off heuristic.

	Board a	Board b	Board c	Total
Win	54%	56%	33%	48 $\pm$ 2 %

Table 6: Win rates of MCTS-PC vs. MCTS-UCT.

Table 6 shows the win rate for MCTS-PC against MCTS-UCT. On boards a and b, 800 games were run to ensure that the observed win rate was not due to random variations. 400 games were run on board c. The bad performance on board c might have to do with the difficulty for a player to isolate itself on this board.

7.3 Expansion Strategy Experiments

In this subsection, experiments are conducted on the predictive expansion (PDE) strategy described in Section 5. The MCTS-PDE program is tested against the MCTS-UCT program. The MCTS-PDE program runs 60,000 play-outs per second on average. The results are shown in Table 7. On each board, 600 games were run.

	Board a	Board b	Board c	Total
Win	53%	58%	48%	53 $\pm$ 2 %

Table 7: Win rates of MCTS-PDE vs. MCTS-UCT.

Similar to the results of the play-out cut-off experiment, the MCTS-PDE program appears to be slightly better than the MCTS-UCT program on board a and b.

The poor win rate on board c is likely caused by the behaviour of the MCTS programs on this board. The programs spiral around the centre, leaving the outer edges of the board open. Since the space estimation heuristic used by the predictive expansion strategy is only applicable when the players are isolated from each other, the MCTS-PDE program is squandering computation time on a mostly useless heuristic. Since the MCTS-UCT program spends all of its time on play-outs, it can look further ahead and therefore has an advantage over the MCTS-PDE program.

7.4 MCTS-Solver Experiments

In this subsection, the Score-Bounded MCTS-Solver is tested. The MCTS-Solver program runs at the same speed as the MCTS-UCT program. In the experiments of MCTS-Solver, MCTS-Solver-PDE and MCTS-Solver-PDE-PC, 400 games were run on each board.

Win	Board a	Board b	Board c	Total
Solver	50%	52%	57%	53 $\pm$ 3 %
Solver-PDE	32%	74%	53%	53 $\pm$ 3 %
Solver-PDE-PC	30%	82%	70%	61 $\pm$ 3 %

Table 8: Win rates of the MCTS-Solver programs vs. MCTS-UCT.

As shown in Table 8, the MCTS-Solver program shows a slight improvement over MCTS-UCT. MCTS-Solver in combination with PDE or PDE-PC performs poorly on board a because tends to cut off one side of the board, and due to lower number of simulations, cannot look ahead far enough to see the resulting outcome (i.e. loss). PDE and PDE-PC perform well on board b and c, probably due to the obstacles and mistakes made by MCTS-UCT.

In terms of computation time and play style, the MCTS-Solver program is the preferred choice since it requires no additional computations once a move has been proven to lead to a guaranteed win (or draw, when a draw is the best achievable outcome). Furthermore, the program can look up and play the shortest move sequence leading to a win by searching for the shortest winning path in the tree.

7.5 Playing against an $\alpha\beta$ program

In this subsection, the MCTS-Solver-Cut program is tested against the winning program of the Tron Google AI Challenge, A1K0N. The A1K0N program uses $\alpha\beta$-search [12] together with a good evaluation function that is primarily based on the tree of chambers heuristic.

	Board a	Board b	Board c	Total
MCTS-UCT	40%	0%	0%	14 ± 3 %
MCTS-PDE	44%	0%	0%	15 ± 3 %
Solver-PDE	13%	12%	0%	8 ± 3 %
Solver-PDE-PC	28%	10%	16%	18 ± 3 %
Solver-PDE-PC-PB	12%	18%	0%	10 ± 3 %

Table 9: Win rates of various MCTS programs against A1K0N.

As can be seen in Table 9, A1K0N is the stronger player by far, achieving a win rate of 82% against MCTS-Solver-PDE-PC and a win rate higher than 85% against the other MCTS programs. Applying PB to the MCTS-Solver program does not seem to give an improvement in overall playing strength.

Although the MCTS program reaches a decent level of play, it still makes mistakes, mainly due to the fact that the reliability of the play-outs rapidly drops as the players get more distant from each other. By the time MCTS sees that it is in a bad position, it is already too late to correct.

8 Conclusion and Future Research

In this paper we developed an MCTS program for the game of Tron. Several enhancements were made to the selection, expansion and play-out phase. All of the enhancements were tested against an MCTS-UCT program.

The enhancement made to the selection phase, the progressive bias strategy, showed no improvement over UCT on two out of three boards. It is unclear why the Progressive Bias enhancement scored a consistent win rate of over 63% on board c.

The experiments of the play-out strategies have shown that the board configuration has a large influence on the game and the effectiveness and accuracy of the play-out strategies. The random-move strategy appeared to be the most reliable choice, doing reasonably well on all three boards. The wall-following strategy outperformed the other strategies only on board b, whereas the defensive strategy outperformed the other strategies on board c.

Applying play-out cut-off showed an increase in playing strength on board a and b (54% and 56%, respectively), but significantly decreased the playing strength on board c. The bad performance on board c may have to do with the difficulty for a player to isolate itself on this board.

Similar to the play-out cut-off enhancement, the predictive expansion strategy showed a slight increase in playing strength on board a (53%) and b (58%), but not on board c. The poor win rate on board c is likely caused by the behaviour of the MCTS programs on this board. The MCTS programs keep a large space open behind them, up until late in the game. On such positions, the space estimation heuristic used is not helpful.

The Score-Bounded MCTS-Solver was tested against MCTS, and turned out to be an improvement, although MCTS-Solver with PDE performs poorly on board a, in comparison to MCTS-PDE. MCTS-Solver is preferred for its ability to look up the shortest path leading to a win (or draw, when a draw is the best achievable outcome) once one or more moves have been proven. In contrast, MCTS-UCT usually postpones the victory as long as possible.

Using PDE on MCTS-Solver enables the program to prove a position more quickly, however, the extra time spent on computing the heuristic did not work out well on all boards. PDE and PC in combination with MCTS-Solver further increased the overall playing strength of the program. MCTS-Solver-PDE-PC achieves a surprisingly high win rate on board c (70%), where MCTS-PC only scored (33%).

The experiment involving the $\alpha\beta$ program showed that the MCTS programs struggle at evaluating positions where the players are distant from one another (further than 10 steps away). Overall, the Solver-PDE-PC program is the best performing program, winning approximately 1 out of 5 games on average against the $\alpha\beta$ program, and achieving a win rate of 61% against MCTS-UCT.

The experiments show that the board configuration

has a large influence on the playing strength of the enhancements tested. Since our goal was to create a Tron program capable of playing on any 13×13 map, the experiments should be conducted on a lot more boards.

As future research, improvement is likely to be gained from further tuning the C constant of MCTS-UCT. Furthermore, MCTS-PB, MCTS-PDE, MCTS-PC and MCTS-Solver and derived programs may have different optimal values for C than MCTS-UCT.

Applying a more sophisticated play-out strategy may increase the playing strength of MCTS in Tron, since even a simple strategy such as the random-move strategy already gives a decent result. It would be interesting to see whether the play-out strategy and selection strategy can be improved such that MCTS can correctly look far ahead. The play-out phase might even have to be replaced completely by a sophisticated evaluation function (e.g. tree of chambers), as used by the $\alpha\beta$ program A1K0N.

References

[1] Arneson, B., Hayward, R.B., and Henderson, P. (2010). Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 251–257.

[2] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine learning*, Vol. 47, No. 2, pp. 235–256.

[3] Berliner, H.J. (1979). The B* Tree Search Algorithm: A Best-first Proof Procedure. *Artificial Intelligence*, Vol. 12, No. 1, pp. 23–40.

[4] Cazenave, T. and Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. *Computers and Games*, Vol. 6515 of *LNCS*, pp. 93–104, Springer.

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

[6] Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)*, Vol. 4630 of *LNCS*, pp. 72–83, Springer-Verlag.

[7] Dmj (2010). Survival Mode. `http://www.ai-contest.com/forums/viewtopic.php?p=1568`.

[8] Fossel, J.D. (2010). Monte-Carlo Tree Search Applied to the Game of Havannah. B.Sc. thesis, Maastricht University.

[9] Gelly, S. and Wang, Y. (2006). Exploration Exploitation in Go: UCT for Monte-Carlo Go. *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*, Citeseer.

[10] Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

[11] Iouri (2010). Survival Mode. `http://www.ai-contest.com/forums/viewtopic.php?p=1484`.

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

[13] Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006*, Vol. 4212 of *LNCS*, pp. 282–293, Springer.

[14] Lorentz, R.J. (2008). Amazons Discover Monte-Carlo. *Computers and Games (CG 2008)*, Vol. 5131 of *LNCS*, pp. 13–24, Springer.

[15] Robbins, H. (1952). Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, Vol. 58, No. 5, pp. 527–535.

[16] Samothrakis, S., Robles, D., and Lucas, S.M. (2010). A UCT Agent for Tron: Initial Investigations. *2010 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 365–371, IEEE.

[17] Sloane, A. (2010). Google AI Challenge Post-mortem. `http://a1k0n.net/blah/archives/2010/03/`.

[18] Sturtevant, N.R. (2008). An analysis of uct in multi-player games. *Computers and Games (CG 2008)*, Vol. 5131 of *LNCS*, pp. 37–49, Springer.

[19] Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA.

[20] University of Waterloo Computer Science Club (2010). Google AI Challenge. `http://csclub.uwaterloo.ca/contest/`.

[21] Voronoi, G. (1907). Nouvelles Applications des Paramètres Continus à la Théorie des Formes Quadratiques. *Journal für die Reine und Angewandte Mathematik*, Vol. 133, pp. 97–178.

[22] Winands, M.H.M., Björnsson, Y., and Saito, J.-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 239–250.