A UCT Agent for Tron: Initial Investigations

Spyridon Samothrakis, David Robles and Simon M. Lucas, *Senior Member, IEEE*

***Abstract*— Monte Carlo Tree Search(MCTS) has generated a great deal of excitement in the A.I. community, mainly due to its success in Go. In this paper we test this approach in Tron, a simultaneous move two-agent game. Although the agents created are able to play to a good standard, there is a degree of randomness in their decisions in identical scenarios. This suggests that the success of MCTS is heavily dependent on the suitability of each individual game for Monte-Carlo Simulations.**

I. Introduction

Throughout the history of Artificial Intelligence (AI), games have provided a popular and challenging platform for research. Traditionally, much of the work on AI in games has centred on board games such as Checkers [1], Chess [2], [3], Backgammon [4] and more recently, Go, the latter being considered now by some researchers as a sort of "drosophila" of AI. Such games usually have well-defined rules which players must stick to, which eases the understanding of the environment and allows one to focus on the creation of AI [1].

With the recent success of Monte-Carlo Tree Search (MCTS) in Computer Go [5], [6] and General Game Playing (GGP) [7], there has been a growing interest in applying these techniques to other board-like strategy games. In this paper we investigate the use of Monte-Carlo Tree Search in the game *Tron*, a Snake-like two player game, where the goal is to box the opponent and make him crash into a wall or his own tail before you. This game was used in the *2010 Google AI Challenge*[2] organised by the University of Waterloo Computer Science Club, which consisted of developing the best agent to play the game using any kind of techniques in a wide selection of programming languages.

From an AI research perspective, there are many aspects of the game worthy of study. It is interesting to see how simulation-based methods perform against traditional game-tree search methods, such as Minimax with Alpha-Beta Pruning. Also, it is important to study the performance of pure Monte-Carlo methods against more refined MCTS techniques, such as the ones using the recently popular UCT algorithms. In addition, in these kind of games the use of heuristics is probably the most fundamental characteristic of the top agents, and since the goal of this work is to study the use of MCTS we see how this techniques perform against agents with very sophisticated heuristics.

The rest of the paper is organised as follows: in Section II we review the game play of Tron, Section III provides the necessary background, Section IV proposes a general methodology for applying UCT, section V describes the experiments and Section VI provides a summary and conclusions.

II. The Game of Tron

The film *Tron* was released in 1982 by Walt Disney Studios. It features a game in a virtual world where two futuristic motorcycles move at constant speed, making only right angle turns and leaving solid wall trails behind them. As the game advances, the arena fills with walls and eventually one opponent dies by crashing into a wall (see figure 1). The game became very popular and was subsequently implemented on various hardware platforms.

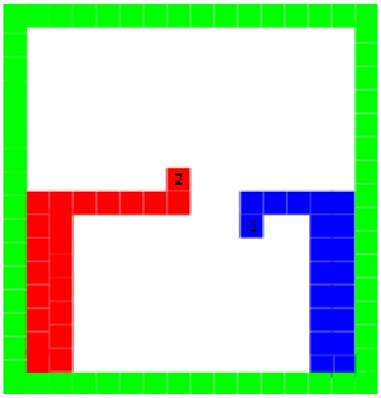

Fig. 1. Tron Game: The players try out-manoeuvre each other

A. Tron Game Play

Tron is played on a M by N grid of cells in which each cell can take two possible states: empty or occupied (wall). The game begins with two players located on empty cells of the grid. At each time step t of the game both players are asked simultaneously to move in any of the four directions: north, south, east or west. If the player moves to an empty cell, that cell becomes a wall. As the game advances, the grid progressively fills with walls and eventually one opponent crashes, thereby ending the game.

To explain in greater detail, the model calls both agents at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t both agents receive a full representation of the model's *state*, $s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states

Spyridon Samothrakis, David Robles and Simon M. Lucas are with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, United Kingdom. (emails: ssamot@essex.ac.uk, darobl@essex.ac.uk, sml@essex.ac.uk).

[1] Videogames are rather different to this, and offer their own challenges: the rules of the game (e.g. the exact physics model used) are not normally known to the players the game objectives are usually not so clearly defined (e.g. for an NPC the objective might be to make the game as fun as possible for the player), and the game worlds are typically much larger and more open-ended.

[2] http://csclub.uwaterloo.ca/contest/

978-1-4244-6297-1/10/$26.00 ©2010 IEEE

and on that basis selects an *action*, $a_{p,t} \in \mathcal{A}_p(s_t)$ where $\mathcal{A}_p(s_t)$ is the set of actions available in state s_t for player p. One time step later, as a consequence of their actions, the agents find themselves in a new state, s_{t+1}. Figure 2 diagrams the agent-model interaction.

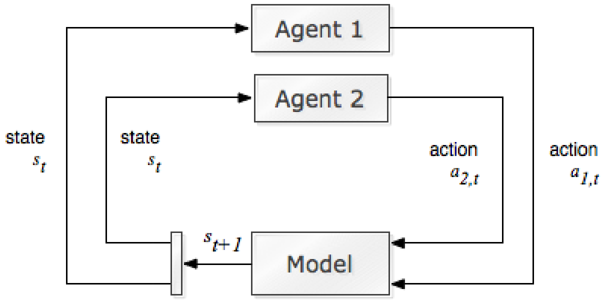

Fig. 2. The agent-model interaction in Tron.

An example of the possible actions that an agent can take in a given situation is depicted in figure 1. In this example the state of the game is on time step 20, s_{20}, in which the agent's action set are:

$\mathcal{A}_1(s_{20}) = \{\texttt{south}, \texttt{east}, \texttt{west}\}$
$\mathcal{A}_2(s_{20}) = \{\texttt{north}, \texttt{east}, \texttt{west}\}$

If one of the agents chooses an action that is not part of the respective set of actions, $\mathcal{A}_p(s_{20})$, the player will crash to a wall and will lead to a loss, or a draw, in the case where the opponent crashed as well.

All the maps used for this research are enclosed by walls, preventing any wrap-round or toroidal grids (i.e. players cannot move and reappear on the opposite side). As might be expected, there are three possible results for a player: win, loss or draw. A player wins when only the opponent crashes against a wall, and draws when both players crash at the same time step. Therefore, the overall general strategy is to try to box the opponent into a small area while your bot can roam the rest of the board. However, to achieve this overall goal more specific strategies must employed.

III. Background

A. Previous Work on Tron

Tron has been used previously as a testbed for research in different domains. Funes et al. [8] used the Internet as a virtual ecosystem, i.e., a community of human users and artificial environments where complex phenomena takes place. They built a coevolutionary environment with a Java implementation of Tron, that matched artificial agents against human Internet users. Using robot vs. robot coevolution in a background server, the fittest artificial agents were selected, which in turn played against human players and kept on evolving. One of the drawbacks of using the humans for fitness was the slow interaction between robots and players, since only a few hundred games took place every day. Also, agents had only limited sensory perception, whereas human players where able to observe the entire state. Apparently this prevented the evolution of better strategies, and it would be quite interesting to see how the agents would have evolved taking into account the entire state of the map.

Later on, Blair et al. [9] used the same coevolutionary system evolving neural network players. The Tron agents used a three-layer neural network with 8 inputs for each sensor, 5 hidden-units, and three outputs for each of the possible moves. The MLP was trained using an evolutionary hill-climbing algorithm in which the top MLP is engaged in a contest by a variety of mutant networks until one defeats it. According to their results, two of the evolved agents displayed common strategies, such as trying to fill the map twisting around a fixed centre point from the edges and progressively moving downward. This strategy decreases the space available for the opponent to move. Also, one of the agents evolved to be extremely aggressive, moving swiftly around the map in an apparently deficient behaviour seeking for opportunities surround the opponent with walls. On the other side, another player evolved to be quite defensive, progressively moving toward the outside in a narrow spiral fashion. Also, another player displayed a versatile behaviour. At the beginning it makes early mistakes, but quickly learns a defensive strategy, and gradually masters an offensive skill to confine its opponent. The top evolved players were tested against a selection of high quality GP agents with a variety of strategies produced in their previous work [8] and seemed to perform relatively well, although it is hard to conclude how effective the strategies are in an objective way.

Funes and Pollack [10] continued their work on [8], by analysing the performance of Tron agents evolving vs. human players with the application of a statistical method similiar to chess and Go "ELO" ratings. This allowed a comparison of all individual players with each other, even when it is possible that they have never played together. The results showed that the difference between the top group of human players and the top agent players is about 60%. Seven out of the best 15 players are agents. They concluded that Tron is partially learnable by self-play, and that a few very good agent players managed to survive. Moreover, the variance in skill level between different evolved top players proved insufficient, as the top agents were not so effective against humans.

B. UCT and Monte Carlo Tree Search

UCT(which stands for Upper Confidence Bound in Trees) is an algorithm from the Monte Carlo Tree Search family of algorithms. The idea behind Monte Carlo algorithms in Artificial intelligence (A.I.) is that the approximation of future rewards (as the are understood in the Markov Decision Process(MDP)[11]) can be achieved through random sampling. What this effectively means is that the agent extrapolates to future states in a random fashion and moves to the state with the highest predicted rewards. MCTS tries to rectify some of the issues that come with such an approach by combining it with a tree and effectively creating a stochastic form of best-first search. From a game theoretic perspective, the tree is a subtree of the game tree in extensive form[12].

UCT (presented in Algorithm 1) takes the ideas of MCTS and pushes one step further by defining the number of Monte Carlo simulations that need to take place before we can say with confidence that we have seen enough from a possible move. Thus, it combines MCTS with ideas borrowed from the multi armed bandit[13]. Each node in the tree is seen as multi-armed bandit problem. The goal of the search is to "push" more towards areas of the search space the seem more promising. Although there are slightly different versions of the algorithm, the one presented in [14] is most commonly used. The algorithm(see figure 3 can been summarised as follows; Starting from the root node, expand the tree for a single node. If the node is a leaf node, give it a value and back-propagate the node value to the nodes that have been rolled out already. If the node is not a tree node, and has not been visited before, perform a Monte Carlo search for this node, and back-propagate as normal. The most commonly used back-propagation strategy is one that makes direct use of the underlying tree, e.g. for a minmax(negamax) tree, that would include subtracting or adding the result depending on who is the owner of the each node in the list(for an example see Algorithms 1,3,4,2, taken from [14]).

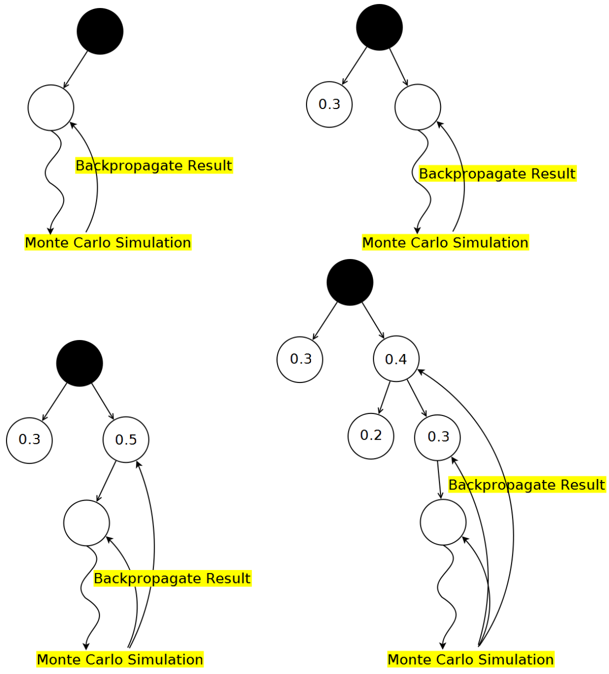

Fig. 3. A sample UCT search

In case all possible nodes have been visited, a common way to distinguish which node to explore further is to assign a value to each node based on the Chernoff-Hoeffding bound. This leads to what is known as the UCB1 policy[13]. Play arm j that maximises

$$\bar{x_j} + C\sqrt{\frac{ln(n)}{n_j}} \tag{1}$$

The symbols $\bar{x_j}$ in equation 1 denotes the average reward from the underlying bandit, and it is the exploitation part of the algorithm. The second part of the above equation is the exploration part. C is a constant (usually set to $\sqrt{2}$)[13], n is the sum of all trials and n_j is the number of trials for the j bandit. Finally, $\sum_{j=0}^{j=j_{max}} n_j = n$.

The equation to choose which arm to play (in the case of a tree search which child to follow) can be heavily tuned depending on the underlying distribution. UCB1(see Algorithm 3 for the pseudocode) should be seen as the "lowest common denominator' policy. If no information about the bandits is available to us, this is probably the correct policy to use. On the other hand, once we have collected enough information, more informed policies are bound to do better.

Algorithm 1 playOneSequence(rootNode)

```
node[0] ← rootNode
i ← 0
repeat
   node[i+1] ← descendByUCB1(node[i])
   $i \leftarrow i + 1$
until node[i] is visited for the first time
createNode(node[i])
node[i].value ← getValueByMC(node[i])
updateValue(node, -node[i].value)
```

Algorithm 2 getValueByMC(node)

```
// perform a random game until completion, return reward.
```

Algorithm 3 descendByUCB1(node)

```
nb ← 0
for i ← 0 to node.childNode.size() - 1 do
   nb ← nb + node.childNode[i].nb
end for
for i ← 0 to node.childNode.size() - 1 do
   if node.childNode[i].nb = 0 then
      v[i] ← $\infty$
   else
      v[i] ← 1.0 - node.childNode[i].value / node.childNode[i].nb + sqrt(2 * log(nb) / (node.childNode[i].nb)
   end if
end for
index ← argmax(v[j])
return node.childNode[index]
```

Note that events in a tree are not independent, so algorithms that would naturally work for bandit problems need some adaptation when applied to tree search. Advantages of

Algorithm 4 updateValue(node, value)

```
nb $\leftarrow$ 0
for i $\leftarrow$ node.size() - 2 to 0 do
    node[i].value $\leftarrow$ node[i].value + value
    node[i].nb $\leftarrow$ node[i].nb + 1
    value $\leftarrow$ 1.0 - value
end for
```

UCT include the fact that it explores the tree asymmetrically and that it gives a natural account for uncertainty.

C. UCT in Games

The popularity of UCT stems primarily from the fact that it revolutionised computer GO[14]. Its success lead to widespread acclaim of Monte Carlo methods, eventually reaching popular media[15]. A key characteristic of all go implementations of UCT ($CRAZYSTONE$[16], $MANGO$[17], $MOGO$[14] and $FUEGO$[18]) is the use of local patterns to guide the Monte Carlo simulations. Gelly et al[14] report a big boost compared to purely random methods(from 1647 to 2200 ELO).

As a result, UCT has been applied already to a big number of games[19], [20], [21]. For the most part, the non-GO papers failed to replicate the burgeoning success of UCT in GO. The area that was identified for improvement[19] was mostly around the concept of doing good Monte Carlo simulations. Being a best-first search, UCT relies heavily on quality of Monte Carlo simulations, and its performance is greatly affected by them.

A big issue with the non-GO implementations of UCT is the lack of comparison with the state of the art. As a consequence there is no way of understanding how well UCT did compared to other methods.

Finally it is worth noting here that UCT is currently very successful in General Game Playing (GGP) [7], a domain that practically prohibits the use of strong heuristics. This is truly important as it signifies that UCT might be used as a generic A.I. technique (at least in the case were a perfect model is available to the agent).

IV. Methodology

A. Applying UCT

There has been no “standard” process for applying UCT, so we are proposing an empirical four step process. The first step is to understand the number of agents and the information content of each game and choose the right tree. For games of complete and perfect information(e.g. chess, GO), a *min-max* tree is commonly used. For games of complete but imperfect information(ex. backgammon), *expectimax* trees should be used. Finally for games of imperfect and incomplete information(ex. poker), *miximax* trees were recently introduced[22].

The second step is understanding the underlying distribution of each arm and tuning the policy equation. This can be done in a number of ways, which can range from tuning equation 1, to completely replacing it with something that captures the underlying probabilities better.

The third step is to come up with a back-propagation policy.

The final step is to augment the algorithm with knowledge and/or ”guide” the Monte Carlo simulations. In Go this is achieved by using local patterns[14], which significantly improves the quality of the simulations.

In this paper we follow a minimalist approach to all these steps, as we would initially like to see the behaviour of MCTS/UCT for playing Tron in a near-default setting.

B. UCT and Tron

Tron lends itself naturally to a min-max version of UCT. Although a simultaneous move game, it can be transformed into an alternate moves games by assuming that in each player’s tree node the current player plays always first. Monte Carlo simulations are easy to implement in this setting. Rewards are provided in a simple fashion, with 1.0 for a victory, 0.5 for a draw an 0.0 for a defeat. Value back-propagation (presented in 3) is simply the mean number for the tree part below.

A very simple form of local search is performed in the Monte Carlo simulation, players cannot play games that result in self-entrapment within one move, as shown in figure 4. This is a minimal form of guiding the searches.

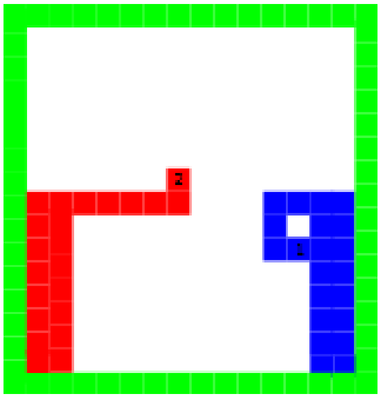

Fig. 4. Player one cannot move north in an MC simulation, a certain lose move

In order to help our agent further, we also switch to “survival mode” when the agents are separated through a wall. The separation of the agents turns the game into single player game similar to “snake”. In this mode, the min-max tree is replaced by a simple tree. In this case, with l being the number of free blocks, the reward function becomes $r = 1/l$. Please note here that finding the longest path is probably an NP-Hard endeavour that necessitates the use of heuristics in very large maps.

V. Experiments

These experiments were designed to gain insight into how “thinking time” (i.e. the time an agent spends for each move) impacts the quality of the agent. In order to achieve this we have devised three different maps (see fig.5). Each map has a different set of qualities that make it suitable for such a task. All maps are of size 15x15. Player one changes with each experiment while player two is the “master player”. The “master player” always uses UCB1 and has one 1 second do think for every move. In our implementation, the uct values in the tree are cached from state to state (i.e. the information gathered from a state through previous simulations is not discarded). In all cases, the move selected is the one that has the higher number of visits (there are alternatives, but it seemed it did not matter in our case).

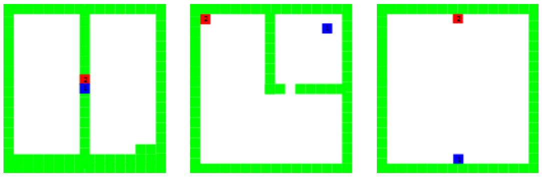

Fig. 5. The three maps used in our experiments, Map A, map B and Map C respectively (left to right)

A. *Maps*

1) Map A: The importance of map A lies in the fact that it should make it hard for a random rewards regime to get any information out of it. While the solution to the problem is obvious to a human player, a random reward agent should have trouble recognising where to move, as it practically has to reach endgame positions with both players playing perfectly before it can see any potential value moving left or right.

TABLE I

PAYOUTS FOR MAP 1

	Player Two Right	Player Two Left
Player One Right	(1,0)	(0,1)
Player One Left	(1,0)	(0.5,0.5)

It is obvious from table I that the correct first move for our agent is moving to the left. It is not the more exploitive move, but it in this case it is the optimal move.

2) Map B: Map B is the equivalent of a chess “checkmate in x moves” problem. In this scenario, player one can always win if it plays right. The goal here is to leave the “cage” that surrounds him and try to actively block player 2.

3) Map C: Map C is the “open field” map and closer to what an agent might encounter towards a real world game.

B. *Agents*

In our experiments we compare three different agents. The first one is configured with with the classic UCB1 policy presented above. For the second one we use a UCB-TUNED[13](see equation 2), which seems to be fairly common in the literature

$$\bar{x_j} + \sqrt{\frac{ln(n)}{n_j} min\left\{1/4, \bar{x_j^2} - \bar{x_j}^2 + \sqrt{\frac{2ln(n)}{n_j}}\right\}} \qquad (2)$$

Our third agent is a modification of the UCB1 strategy presented by Coquelin, P.A. and Munos, R.[23]. We call this strategy UCB-E.

$$\bar{x_j} + C\sqrt{\frac{\sqrt{n}}{n_j}} \qquad (3)$$

The above equation 3 pushes towards more exploration

C. *Experiment 1: UCB1*

In this first experiment we compare agents based on UCB1 with each other. For each of the maps with play 20 games between our “master player” and test players. The Master player is the disadvantaged player in Map 2. It’s thinking time is 1000ms. The test players are identical to the master player, but their calculating time varies from 100ms to 900ms, with 50ms increments. The score accumulated for each player is $s = 0.5 * nDraws + nWins$.

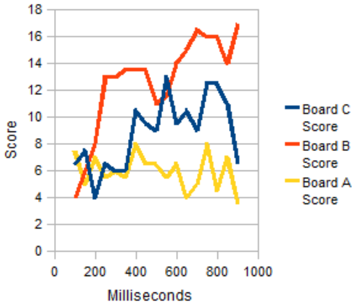

Fig. 6. Score of agent UCB1 for each map

The results in figure 6 are somewhat surprising. While in map B can see that the best test player wins 17/20 times, it is obvious that the agent cannot achieve perfect performance. Results for Map A and C show a random trend, with no clear advantages as more thinking time is provided to the agent.

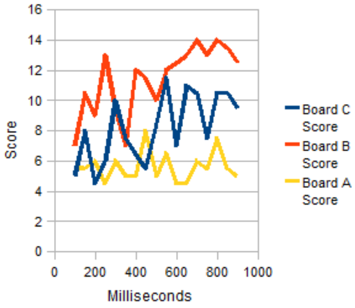

Fig. 7. Scores of agent UCB-TUNED for each map

D. *Experiment 2: UCB-TUNED*

The UCB-TUNED strategy has been used in GO[14] and OTHELLO[20]. Overall it performs slightly worse than UCB1. This strategy is overall more exploitive/optimistic than UCB1, which seams to hinder its ability to search for good solutions on Map B(see figure 7). In the other hand, it seems to be doing better (although this is debatable without further statistical analysis as the results are too close) on Map A, possibly because it reaches deeper nodes in the tree.

E. Experiment 3: UCB-E

This strategy pushes towards more exploration. It is affected more by the thinking time than the other two strategies, as it explores the tree more thoroughly.

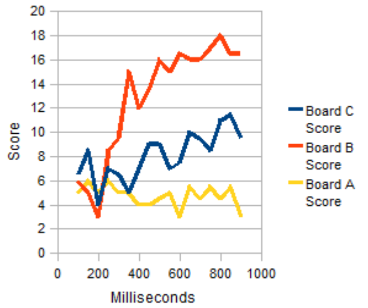

Fig. 8. Scores of agent UCB-E for each map

In figure 8 we can see that it starts off somewhat worse than the other two strategies, but as more thinking time is provided, it seems to do as well as the other two. The shortest tree search (less exploitation) seems to have trouble doing well on Map A,were reaching deep in the tree is crucial.

F. Experiment 4: 10s UCB1 Opponent

Finally, we used the most successful strategy from above for doing a final run. We let the UCB1 agent peform for 10 seconds against the "master" player used in the previous examples. The scores are not impressive (although somewhat better, see figure 9), meaning that although there is an improvement the 900ms agent is close to convergence. The scores for this agent can be found in table II.

TABLE II

SCORES FOR 10 SECONDS PLAYER

	Board A Score	Board B Score	Board C Score
UCB1	8.5	19.5	12.5

G. Experiment 5: Tron Competition

A slightly modified version of UCB1 has competed in the U of Waterloo Tron Competition, achieving a rank of 109/750(approx). The winning entries were based mostly on min-max with excellent leaf evaluation functions[3]. Monte Carlo approaches overall did poorly, with positions ranging from 25 to 200. As far as the authors are aware the best UCT approach removed the Monte Carlo simulations completely and used a heuristic function instead, while keeping the best first search UCT exploration of the tree[4]. This arguably goes against the "spirit" of MCTS but it might prove to be the right path if there is no effective way of guiding the Monte Carlo simulations.

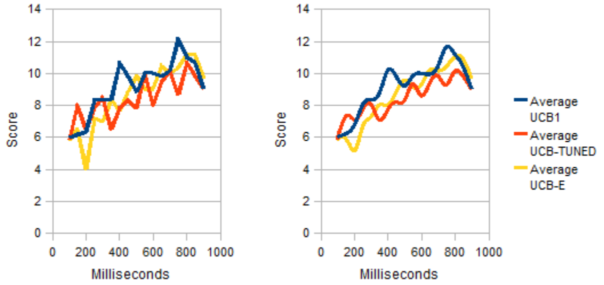

Fig. 9. A comparison of the averages for each UCT heuristic alongside its smoothed version

VI. Conclusions

MCTS/UCT is attracting significant attention from the AI and games research community. It is therefore of great interest to gain a better understanding of when it succeeds and fails and for what reasons. We compared three different variations of UCT using three different carefully chosen boards in the Game of Tron. From the results one can clearly see that UCT works sufficiently well but its straightforward version is heavily dependent on the quality of the Monte-Carlo simulations. In some cases a large number of simulations are irrelevant, pointing UCT to explore completely ineffective branches. An interesting future research direction would be to try to extract features from the game, and use them to guide the search.

Acknowledgements

This research is partly funded by a postgraduate studentship from the Engineering and Physical Sciences Research Council. We would also like to thank the University of Waterloo computer science club for setting up the Tron Competition.

References

[1] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal on Research and Development*, vol. 11, no. 6, pp. 601–617, 1967.

[2] C. Shannon, "Programming a computer for playing chess," *Philosophical Magazine*, vol. 41, no. 4, pp. 256–275, 1950.

[3] M. Campbell, A. J. Hoane, Jr., and F. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[4] G. Tesauro, "Temporal difference learning and TD-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[5] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Proceedings of the 5th International Conference on Computers and Games (CG2006)*, 2006, pp. 72–83.

[6] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modifications of uct with patterns in monte-carlo go," INRIA, Tech. Rep. 6062, 2006.

[3] `http://www.a1k0n.net/blah/`

[4] `http://csclub.uwaterloo.ca/contest/forums/viewtopic.php/?f=8&t=361&start=10`

[7] Y. Bjornsson and H. Finnsson, “Cadiaplayer: A simulation-based general game player,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, pp. 4–15, 2009.

[8] P. Funes, E. Sklar, H. Juillé, and J. Pollack, “Animal-animat coevolution: Using the animal population as fitness function,” in *Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, 1998, pp. 525–533.

[9] A. D. Blair, E. Sklar, and P. Funes, “Co-evolution, determinism and robustness,” in *SEAL'98: Selected papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning*, 1999.

[10] P. Funes and J. Pollack, “Measuring progress in coevolutionary competition,” in *Proceedings of the Sixth International Conference on the Simulation of Adaptive Behavior*, 2000, pp. 450–459.

[11] L. Kocsis and C. Szepesvári, “Bandit based monte-carlo planning,” in *15th European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.

[12] E. Rasmusen, *Games and information: An introduction to game theory*. Blackwell Pub, 2007.

[13] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[14] S. Gelly and Y. Wang, “Exploration exploitation in go: UCT for Monte-Carlo go,” in *Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006)*. Citeseer, 2006.

[15] R. Blincoe, “Go, going, gone?” The Guardian, 2006. [Online]. Available: http://www.guardian.co.uk/technology/2009/apr/30/games-software-mogo/print

[16] R. Coulom, “Computing elo ratings of move patterns in the game of go,” in *Computer Games Workshop*. Citeseer, 2007.

[17] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, and B. Bouzy, “Progressive strategies for monte-carlo tree search,” *New Mathematics and Natural Computation*, vol. 4, no. 3, p. 343, 2008.

[18] “Fuego - an open-source framework for board games and go engine based on monte-carlo tree search,” University of Alberta, Dept. of Computing Science, TR09-08, Tech. Rep., April 2009. [Online]. Available: http://www.cs.ualberta.ca/TechReports/2009/TR09-08/TR09-08.pdf

[19] F. Van Lishout, G. Chaslot, and J. Uiterwijk, “Monte-Carlo Tree Search in Backgammon,” in *Computer Games Workshop*, 2007, pp. 175–184.

[20] P. Hingston and M. Masek, “Experiments with Monte Carlo Othello,” in *IEEE Congress on Evolutionary Computation, 2007. CEC 2007*, 2007, pp. 4059–4064.

[21] G. V. den Broeck, K. Driessens, and J. Ramon, “Monte-carlo tree search in poker using expected reward distributions,” in *ACML*, 2009, pp. 367–381.

[22] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron, “Game-tree search with adaptation in stochastic imperfect-information games,” *Lecture Notes in Computer Science*, vol. 3846, pp. 21–34, 2006.

[23] P. Coquelin and R. Munos, “Bandit algorithms for tree search,” *Arxiv preprint cs/0703062*, 2007.