Monte-Carlo Tree Search for the Physical Travelling Salesman Problem

Diego Perez, Philipp Rohlfshagen, and Simon M. Lucas

School of Computer Science and Electronic Engineering
University of Essex, Colchester CO4 3SQ, United Kingdom
{dperez, prohlf, sml}@essex.ac.uk

Abstract. The significant success of MCTS in recent years, particularly in the game Go, has led to the application of MCTS to numerous other domains. In an ongoing effort to better understand the performance of MCTS in open-ended real-time video games, we apply MCTS to the Physical Travelling Salesman Problem (PTSP). We discuss different approaches to tailor MCTS to this particular problem domain and subsequently identify and attempt to overcome some of the apparent shortcomings. Results show that suitable heuristics can boost the performance of MCTS significantly in this domain. However, visualisations of the search indicate that MCTS is currently seeking solutions in a rather greedy manner, and coercing it to balance short term and long term constraints for the PTSP remains an open problem.

1 Introduction

Games such as Chess have always been a popular testbed in the field of Artificial Intelligence to prototype, evaluate and compare novel techniques. The majority of games considered in the literature are two-player turn-taking zero-sum games of perfect information, though in recent years the study of AI for video game agents has seen a sharp increase. The standard approach to the former type of game is *minimax* with $\alpha\beta$ pruning which consistently chooses moves that maximise minimum gain by assuming the best possible opponent. This technique is optimal given a complete game tree, but in practice needs to be approximated given time and memory constraints: a value function may be used to evaluate nodes at depth d. The overall performance of $\alpha\beta$ depends strictly on the quality of the value function used. This poses problems in games such as Go where a reliable state value function has been impossible to derive to date. It is possible to approximate the minimax tree, without the need for a heuristic, using Monte Carlo Tree Search (MCTS), though in practice MCTS still benefits significantly from good heuristics in most games.

MCTS is a best-first tree search algorithm that incrementally builds an asymmetric tree by adding a single node at a time, estimating its game-theoretic value by using self-play from the state of the node to the end of the game: each iteration starts from the root and descends the tree using a *tree policy* until a leaf node has been reached. The simulated game is then continued along a previously unvisited state, which is subsequently added to the tree, using the *default policy* until the end of the game. The actual outcome of the game is then back-propagated and used by the tree policy in subsequent roll-outs.

The most popular tree policy π_T is UCB1, based on upper confidence bounds for bandit problems [7]:

$$\pi_T(s_i) = \underset{a_i \in A(s_t)}{\arg\max} \left\{ Q(s_t,\ a_i) + K\sqrt{\frac{\log N(s_t)}{N(s_t, a_i)}} \right\} \quad (1)$$

where $N(s_t)$ is the number of times node s_t has been visited, $N(s_t,\ a_i)$ the number of times child i of node s_t has been visited and $Q(s_t,\ a_i)$ is the expected reward of that state. K is a constant that balances between exploitation (left-hand term of the equation) and exploration (right-hand term). MCTS using UCB1 is generally known as UCT. In the simplest case, the default policy π_D is uniformly random: $\pi_D(s_t) = rand(A(s_t))$.

In this paper we study the performance of MCTS on the single-player real-time Physical Travelling Salesman Problem (PTSP). This study is part of an ongoing effort that explores the applicability of MCTS in the domain of real-time video games: the PTSP, where one has to visit n cities as quickly as possible by driving a simple point-mass, provides an excellent case study to better understand the strengths and weaknesses of MCTS in open-ended[1] and time-constrained domains. The PTSP also has the feature that the best possible score for a map is usually unknown. Hence, even assigning a value to a roll-out that does terminate (cause the salesman to visit all cities) raises interesting issues. The experimental studies presented in this paper offer new insights into the behaviour of MCTS in light of these attributes and may be used to create stronger AI players for more complex video games in the future, or perhaps more importantly, create intelligent players with very little game-specific programming required.

2 Literature Review

MCTS was first proposed in 2006 (see [3,4]) and rapidly became popular due to its significant success in computer Go [6], where traditional approaches had been failing to outplay experienced human players. MCTS has since been applied to numerous other games, including games of uncertain information and general game playing. In this section we review applications of MCTS to domains most closely related to the PTSP, including optimisation problems, single-player games (puzzles) and real-time strategy games.

MCTS and other Monte Carlo (MC) methods have been applied to numerous combinatorial optimisation problems, including variations of the classical Travelling Salesman Problem (TSP). For instance, Rimmel et al. [9] used a nested MC algorithm to solve the TSP with time windows, reaching state of the art solutions in problems with no more than 29 cities. Bnaya et al. [2] obtained near-optimal results using UCT to solve the Canadian Traveller Problem, a variation of the TSP where some edges of the graph might be blocked with some probability.

Related to optimisation problems are single player games, also known as puzzles. The Single-Player MCTS (SP-MCTS) was introduced by Shadd et al. [13], where a

[1] Open-ended in this context means that many lines of play will never terminate.

modification of UCT was proposed in order to include the effect of not having an opponent to play against. The authors found that restarting the seed of the random simulations periodically, while saving the best solution found so far, increases the performance of the algorithm for single player games. Another puzzle, SameGame, has also been addressed by many researchers, including Schadd et al. [13]. They used SP-MCTS with modified back propagation, parameter tuning and a meta-search extension, to obtain the highest score ever obtained by any AI player so far. Matsumoto et al. [8] incorporated domain knowledge to guide the MC roll-outs, obtaining better results with a little more computational effort. Similarly, Björnsson and Finnsson [1] proposed a modification of standard UCT in order to include the best results of the simulations (in addition to average results) to drive the search towards the most promising regions. They also stored good lines of play found during the search which may be used effectively in single-player games.

MCTS has also been applied widely to *real-time* games. The game Tron has been a benchmark for MCTS in several studies. Samothrakis et al. [11] apply a standard implementation of MCTS, including knowledge to avoid self-entrapment in the MC roll-outs. The authors found that although MCTS works well a significant number of random roll-outs produce meaningless outcomes due to ineffective play. Den Teuling [5] applied UCT to Tron with some modifications, such as progressive bias, simultaneous moves, game-specific simulation policies and heuristics to predict the score of the game without running a complete simulation. The enhancements proposed produce better results only in certain situations, depending on the board layout. Another real-time game that has frequently been considered by researchers is Ms. Pac-Man. Robles et al. [10] expand a tree with the possible moves that Ms. Pac-Man can perform, evaluating the best moves with hand-coded heuristics, and a flat MC approach for the end game prediction. Finally, Samothrakis et al. [12] used MCTS with a 5-player max^n game tree, where each ghost is treated as an individual player. The authors show how domain knowledge produced smaller trees and more accurate predictions during the simulations.

3 The Physical Travelling Salesman Problem

The Physical Travelling Salesman Problem (PTSP) is an extension of the Travelling Salesman Problem (TSP). The TSP is a very well known combinatorial optimisation problem in which a salesperson has to visit n cities exactly once using the shortest route possible, returning to the starting point at the end. The PTSP converts the TSP into a single-player game and was first introduced as a competition at the Genetic and Evolutionary Computation Conference (GECCO) in 2005. In the PTSP, the player always starts in the centre of the map and cities are usually distributed uniformly at random within some rectangular area; the map itself is unbounded.

Although the original PTSP challenge was not time constrained, the goal of the current PTSP is to find the best solution in real-time. At each game tick the agent selects one of five force vectors to be applied to accelerate a point mass around the map with the aim of visiting all cities. The optimality of the route is the time taken to traverse it which differs from its distance as the point-mass may travel at different speeds. At any moment in time, a total of 5 actions may be taken: forward, backward, left, right and

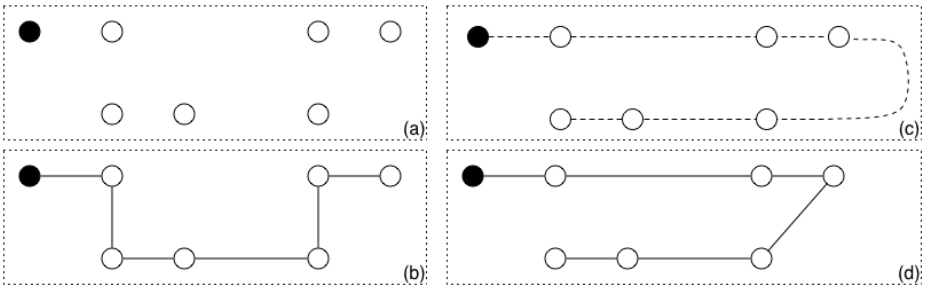

Fig. 1: Example of a 6 city problem where the optimal TSP route differs from the optimal PTSP route: (a) the six cities and starting point (black circle); (b) the optimal TSP solution to this problem without returning to the start; (c) optimal PTSP route and (d) equivalent TSP route which is worse than the route shown in (b).

neutral. At each time step, the position and velocity of the point-mass is updated using Newton's equations for movement: $v = v_i + a\Delta t$ and $s = s_i + v_i\Delta t + \frac{1}{2}a(\Delta t)^2$ with $\Delta t = \sqrt{0.1}$.

There are at least two high-level approaches to confront this problem: one possibility is to address the order of cities and the navigation (steering) of the point mass independently. However, it is important to keep in mind that the physics of the game make the PTSP quite different from the TSP. In particular, the optimal order of cities for a given map which solves the TSP does not usually correspond to the optimal set of forces that can be followed by an agent in the PTSP. This is illustrated in Figure 1. Another possible approach to tackle the PTSP is thus to attempt to determine the optimal set of forces and order of cities simultaneously.

The PTSP can be seen as an abstract representation of video games characterised by two game elements: order selection, and steering. Examples of such games include CrystalQuest, XQuest and Crazy Taxi. In particular, the PTSP has numerous interesting attributes that are commonly found in these games: players are required to act quickly as the game progresses at every time step. Furthermore, the game is open-ended as the point-mass may travel across an unbounded map – it is thus highly unlikely that MCTS with a uniform random default policy would be able to reach a terminal state. This requires the algorithm to (a) limit the number of actions to take on each roll-out (depth); and (b) implement a value function that scores each state. Finally, it is important to note that there is no win/lose outcome which affects the value of K for the UCB1 policy (see Equation 1).

4 Preliminary Experimental Study

The application of MCTS to the PTSP requires a well-defined set of states, a set of actions to take for each of those states and a value function that indicates the quality of each state. Each state is uniquely described by the position of the point-mass, its velocity, and the minimum distance ever obtained to all cities. The actions to take are identical

across all states (forward, backward, left, right and neutral). Finally, the value (fitness) function used is the summation of the values v_i, based on the minimum distances ever obtained to all cities, plus some penalty due to travelling outside the boundaries of the map. The number of steps is also considered. The value v_i is calculated for each city as follows:

$$v_i = \begin{cases} 0 & \text{if } d_i < c_r \\ f_m - \frac{f_m}{d_i - c_r + 2} & \text{otherwise} \end{cases} \quad (2)$$

where d_i represents the distance between the point-mass and the city i, c_r is the radius of each city and f_m is the maximum value estimated for the fitness. This equation forces the algorithm to place more emphasis on the positions in the map that are very close to the cities. The score associated with each state is normalised: the maximum fitness is equivalent to the number of cities multiplied by f_m, plus the penalties for travelling outside the boundaries of the map and the number of steps performed so far. The normalisation is very important to identify useful values of K which has been set to 0.25 in this study following systematic trial and error.

Two default policies have been tested. The first uses uniform random action selection while the second, DRIVEHEURISTIC, includes some domain knowledge to bias move selection: it penalises actions that do not take the agent closer to any of the unvisited cities. This implies that actions which minimise the fitness value are more likely to be selected. Four algorithms are considered in this preliminary experiment: the simplest is 1-ply MC Search which uses uniform random action selection for each of the actions available from the current state, selecting the best one greedily. A slight modification of this is Heuristic MC which biases the simulations using the DRIVEHEURISTIC. The first MCTS variant is using UCB1 as tree policy and uniform random rollouts. The heuristic MCTS implementation also uses UCB1 as its tree policy, but biases the rollouts using the DRIVEHEURISTIC.

To compare the different configurations, the following experiments have been carried out on 30 different maps that have been constructed uniformly at random. A minimum distance between cities prevents overlap. The same set of 30 maps has been used for all experiments and configurations were tested over a total of 300 runs (10 runs per map). Finally, the time to make a decision at each stage of the problem has been set to 10 milliseconds.

The results are shown in Table 1.[2] It is evident that MCTS outperforms, with respect to the number of best solutions found, both 1-ply MC Search and MCTS with a heuristic in the roll-outs. The differences in the average scores are, however, insignificant. The observation that 1-ply MC Search is achieving similar results to MCTS suggests that the information obtained by the roll-outs is either not utilised efficiently or is simply not informative enough. If this is the case, the action selection and/or tree selection mechanism cannot be effective.

Figures 2 and 3 depict a visualisation of the MCTS tree at a particular iteration: each position, represented in the figure as a pixel on the map, is drawn using a grey scale that represents how often this position has been occupied by the point-mass during the MC simulations. Lighter colours indicate more presence of those positions in the simulations, while darker ones represent positions less utilised. Figure 3 is of a special

[2] The experiments were executed on an Intel Core i5 PC, with 2.90GHz and 4GB of memory.

Algorithm	Average	Standard Error	Best count	Not solved	Average simulations
1-ply Monte Carlo Search	539.65	3.295	0	1	816
Heuristic MC	532.63	3.302	3	0	726
MCTS UCB1	531.25	3.522	10	2	878
Heuristic MCTS UCB1	528.77	3.672	2	0	652

Table 1: Results for 10 city maps compared in terms of the number of time steps required to solve the problem. The *Best count* indicates how often the algorithm produced the overall best solution, while *Not solved* shows the number of trials where the algorithm was unable to find a solution. Finally, *Average simulations* indicates how many simulations were performed, on average, in each execution step.

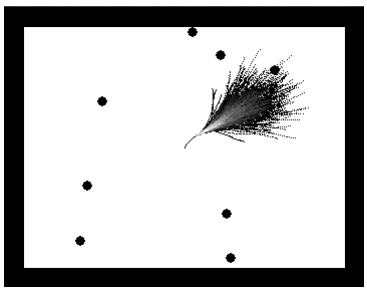

Fig. 2: Tree exploration at the start.

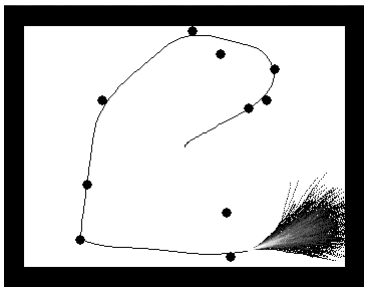

Fig. 3: No cities close to tree exploration

interest: the simulations are taking the point-mass to positions where no cities can be found. This happens because no other portion of the map (where cities are located) is explored and thus MCTS is unable to steer the point-mass towards regions of higher fitness.

5 Extended Experimental Study

The objective of the extended experimental study is to analyse the impact of additional domain knowledge on the performances of the algorithms. In particular, we exploit the concept of a *centroid*, calculated as the centre of all unvisited cities (and the centre of the map). The idea is to prevent the selection of actions that take the point-mass away from that point. However, this penalty cannot be used all the time, because otherwise the point-mass would always be drawn towards the centroid, not visiting any cities. In order to decide when this heuristic is to be enabled, we define a circle with a certain radius *r*, centred on the centroid, to be the centroid's influence.

The maps depicted in Figures 4 and 5 show a set of cities, the centroid (located near the center of the map) and the centroid's area of influence.[3] The value of *r* used is the

[3] Videos of the tree and the *CentroidHeuristic* may be found at `www.youtube.com/user/MonteCarloTreeSearch`.

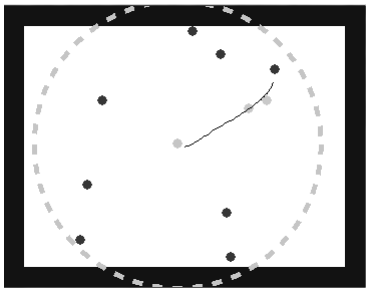

Fig. 4: Centroid and influence.

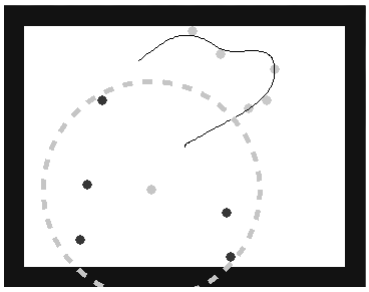

Fig. 5: Centroid and influence update.

distance to the farthest city from the centroid, multiplied by a factor *e*; the value of *e* can be used to modulate how far the point-mass is allowed to go from the centroid. In this study the value is set to 1.05.

We define the following new algorithms using the CENTROIDHEURISTIC: the *Centroid Heuristic MC* is identical to the *Heuristic MC* but the heuristic used to guide the MC simulations ignores actions that do not take the point-mass towards the centroid (if within the centroid influence). Likewise, the *Centroid Heuristic MCTS* is similar to the *Heuristic MCTS* but uses the *CentroidHeuristic* during the default policy. The *Centroid MCTS only UCT* is identical to the standard *MCTS* algorithm but uses the *Centroid-Heuristic* in the tree policy, by not allowing the selection of those actions that do not take the point-mass towards the centroid (if within the centroid influence). Finally, the *Centroid MCTS & UCT* is similar to the *Centroid Heuristic MCTS only UCT* using the *CentroidHeuristic* also in the default policy.

5.1 Random Maps of 10 Cities

The results for random maps of 10 cities are shown in Table 2. It is evident that solution quality was improved by the CENTROIDHEURISTIC. The average solution quality, using the centroid heuristic for both the tree selection and MC roll-outs, is 481.85, with a low standard error and a very good count of best solutions found: both *Centroid MCTS only UCT* and *Centroid MCTS & UCT*, with $K = 0.05$, achieve more than the 50% of the best scores. The Kolmogorov-Smirnov test confirms that these results are significant. The test provides a *p-value* of 1.98×10^{-22} when comparing *1-ply MC Search* and *Centroid MCTS only UCT*, and 1.98×10^{-22} for *1-ply Monte Carlo Search* against *Centroid MCTS & UCT*.

Similar results have been obtained for time steps of 50 milliseconds. In this case, the *1-ply Monte Carlo Search* algorithm achieves an average time of 514.31, while *MCTS UCB1*, *Centroid MCTS only UCT* and *Centroid MCTS & UCT* obtain 511.87, 556.53 and 469.72 respectively. Several things are worth noting from these results: first, the algorithms perform better when the time for simulations is increased. Second, it is interesting to see how the different MCTS configurations (specially *MCTS UCB1* and *Centroid MCTS only UCT*) improve more than the MC techniques when going from 10 to 50ms. The third MCTS configuration, which obtains the best results for

Algorithm	Average	Standard Error	Best count	Not solved	Average simulations
1-ply Monte Carlo Search	539.65	3.295	1	1	816
Heuristic MC	532.63	3.302	1	0	726
MCTS UCB1	531.25	3.522	2	2	878
Heuristic MCTS UCB1	528.77	3.672	1	0	652
Centroid Heuristic MC	552.87	4.158	2	0	915
Centroid Heuristic MCTS	524.13	3.443	2	0	854
Centroid MCTS only UCT	599.38	10.680	6	76	1009
Centroid MCTS & UCT	**481.85**	6.524	**12**	0	659

Table 2: 10 city result comparison, 10ms limit.

both time limits, does not improve its solution quality as much as the other algorithms when increasing the simulation time. However, it is important to note that the results obtained by this configuration given 10ms are better than the best solution found by any other algorithm given 50ms. It is highly significant that the solutions obtained by this algorithm are the best ones found for this problem, showing an impressive performance even when the available time is very limited. This makes the approach very suitable for time-constrained real-time games.

5.2 Random Maps of 30 Cities

To check if the results of the previous section are consistent, some experiments were performed with 30 cities. Table 3 shows the results of these algorithms for a time limit of 10ms. The results are similar to the ones recorded in the 10 cities experiments, although in this case *Centroid MCTS only UCT* with $K = 0.05$ is the algorithm that solves the problem in the least number of time steps. Figure 6 shows the performance of some of the configurations tested for the different time limits considered. Comparing *1-ply Monte Carlo Search* with the *Centroid MCTS only UCT* using the Kolmogorov-Smirnov test gives the following p-values for 10ms, 20ms and 50ms respectively: 0.349, 0.0005 and 1.831×10^{-7}. This confirms significance in the case of 20ms and 50ms, but not in the case of 10ms.

6 Conclusions

This paper analyses the performance of Monte Carlo Tree Search (MCTS) on the Physical Travelling Salesman Problem (PTSP), a real-time single player game. The two experimental studies outlined in this paper focus on the impact of domain knowledge on the performance of the algorithms investigated and highlight how a good heuristic can significantly impact the success rate of an algorithm when the time to select a move is very limited.

The results show that the *CentroidHeuristic* helps the algorithm to find better solutions, especially when the time allowed is very small (10ms). As shown in the results,

Algorithm	Average	Standard Error	Best count	Not solved	Average simulations
1-ply Monte Carlo Search	1057.01	6.449	2	0	562
Heuristic MC	1133.10	6.581	0	11	319
MCTS UCB1	1049.16	6.246	6	0	501
Heuristic MCTS UCB1	1105.46	5.727	2	3	302
Centroid Heuristic MC	1119.93	6.862	0	1	441
Centroid Heuristic MCTS	1078.51	5.976	1	0	428
Centroid MCTS only UCT	**1032.94**	6.365	**7**	14	481
Centroid MCTS & UCT	1070.86	6.684	**7**	0	418

Table 3: 30 city result comparison, 10ms limit.

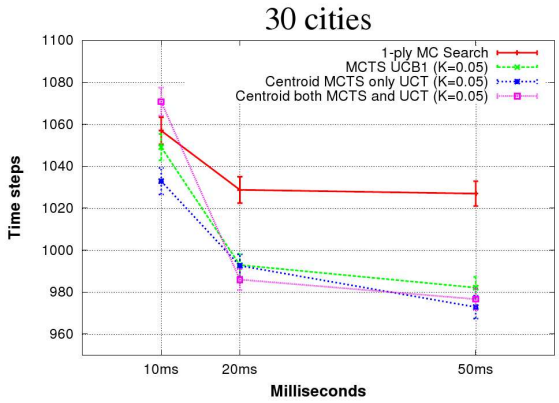

Fig. 6: Performance of the algorithms when time limit changes (30 cities).

when the time limit is 10ms, some approaches (like MCTS without domain knowledge) are not able to provide significantly better results than 1-ply Monte Carlo Search. They are, however, able to produce superior results when the time limit is increased. The main contribution of this research is evidence to show that it is possible to effectively utilise simple domain knowledge to produce acceptable solutions, even when the time to compute the next move is heavily constrained.

The off-line version of the problem has also been solved with evolutionary algorithms, notably in the GECCO 2005 PTSP Competition. In fact, the winner of that competition utilised a genetic algorithm, using a string with the five available forces as a genome for the individuals (results and algorithms employed can be found at `cswww.essex.ac.uk/staff/sml/gecco/ptsp/Results.html`). This PTSP solution format, a string of forces, is a suitable representation for evolutionary algorithms that may be applied to this problem. A thorough comparison of evolution versus MCTS for this problem would be interesting future work.

Other ongoing work includes the use of more interesting maps (by introducing obstacles, for instance), modified game-physics to steer a vehicle rather than a point-mass, and the inclusion of more players that compete for the cities. Competitions based on these variations are already in preparation. The results of these should provide further insight into how best to apply MCTS to the PTSP, as well as its strengths and weaknesses compared to evolutionary and other optimisation methods.

Finally, a particular challenge for MCTS applied to the PTSP is how to persuade it to make more meaningful simulations that consider the long-term plan of the order in which to visit the cities, together with the short term plan of how best to steer to the next city or two.

Acknowledgements

This work was supported by EPSRC grant EP / H048588 / 1.

References

1. Y. Björnsson and H. Finnsson, "CadiaPlayer: A Simulation-Based General Game Player," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 4–15, 2009.
2. Z. Bnaya, A. Felner, S. E. Shimony, D. Fried, and O. Maksin, "Repeated-task Canadian traveler problem," in *Proceedings of the International Symposium on Combinatorial Search*, 2011, pp. 24–30.
3. G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI," in *Proc. of the Artificial Intelligence for Interactive Digital Entertainment Conference*, 2006, pp. 216–217.
4. R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proc. of the 5th Int. Conference on Computer Games*. Springer-Verlag, 2006, pp. 72–83.
5. N. G. P. Den Teuling, "Monte-Carlo Tree Search for the Simultaneous Move Game Tron," Univ. Maastricht, Tech. Rep., 2011.
6. S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.
7. L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," *Machine Learning: ECML 2006*, pp. 282–293, 2006.
8. S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo, and H. Futahashi, "Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem," in *Proc. of the International Multi Conference of Engineers and Computer Scientists*, vol. 3, 2010, pp. 2086–2091.
9. A. Rimmel, F. Teytaud, and T. Cazenave, "Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows," in *Proc. of EvoApplications 2, LNCS 6625*, 2011, pp. 501–510.
10. D. Robles and S. M. Lucas, "A Simple Tree Search Method for Playing Ms. Pac-Man," in *Proc. of the IEEE Conference on Computational Intelligence and Games*, 2009, pp. 249–255.
11. S. Samothrakis, D. Robles, and S. M. Lucas, "A UCT Agent for Tron: Initial Investigations," in *Proc. of IEEE Conference on Computational Intelligence and Games*, 2010, pp. 365–371.
12. ——, "Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man," *IEEE Trans. on Computational Intelligence and AI in Games*, vol. 3, no. 2, pp. 142–154, 2011.
13. M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search," in *Proc. of Computer Games, LNCS 5131*, 2008, pp. 1–12.