Monte Carlo Tree Search: Long-term versus Short-term Planning

Diego Perez, Philipp Rohlfshagen, *Member, IEEE*, Simon M. Lucas, *Senior Member, IEEE*

***Abstract*—In this paper we investigate the use of Monte Carlo Tree Search (MCTS) on the Physical Travelling Salesman Problem (PTSP), a real-time game where the player navigates a ship across a map full of obstacles in order to visit a series of waypoints as quickly as possible. In particular, we assess the algorithm's ability to plan ahead and subsequently solve the two major constituents of the PTSP: the order of waypoints (long-term planning) and driving the ship (short-term planning). We show that MCTS can provide better results when these problems are treated separately: the optimal order of cities is found using Branch & Bound and the ship is navigated to collect the waypoints using MCTS. We also demonstrate that the physics of the PTSP game impose a challenge regarding the optimal order of cities and propose a solution that obtains better results than following the TSP route of minimum Euclidean distance.**

I. Introduction

Games haven always been a popular benchmark for testing new techniques in computational intelligence. Real-time (video) games have become increasingly popular in recent years and many competitions are held at international conferences every year where competitors from different areas of research compete to be the best. Video games tend to be very complex and players must solve a wide range of problems, often in very little time, to make progress. To better understand these requirements, it is useful to examine some of the characteristics of such games in a simplified framework.

In this paper we focus on a simple single-player real-time game: the Physical Travelling Salesman Problem (PTSP) requires the player to navigate a ship in real-time across a map filled with obstacles to collect a series of waypoints as quickly as possible. Despite its simplicity, the PTSP is representative of the numerous challenges a player faces in more complex video games, such as real-time constraints, continuous state spaces and open-endedness. The PTSP lacks the presence of an opponent and hence one is able to plan ahead without having to worry about the actions carried out by the adversary. However, this does not make it trivial: the PTSP is a real-time game where the action to execute must be chosen quickly. Hence, it is usually not possible to plan the entire gameplay at the early stages of the game. Furthermore, the search space may simply be too big to perform deep searches. In many cases one has to choose the best of the options currently available, recomputing moves in the future as required.

Diego Perez (dperez@essex.ac.uk), Philipp Rohlfshagen (prohlf@essex.ac.uk) and Simon M. Lucas (sml@essex.ac.uk), School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK.

The remainder of this paper is structured as follows: Section II highlights the challenges of long-term planning in real-time games when applying Monte Carlo (MC) methods. Section III introduces MCTS for single-player real-time games, followed by the problem description in Section IV. Section V presents the experimental study. Finally, conclusions and future work are discussed in Section VI.

II. Long term vs. Short term planning

Monte Carlo (MC) methods, at least those that sample from a uniform distribution, may be unsuitable for long-term planning: the random nature of the sampling usually lacks sufficient direction to explore the search space in a satisfactory manner. In most video games, MC simulations are not able to reach terminal states as this usually requires thousands of moves. Monte Carlo Tree Search (MCTS) is able to look ahead further by building an asymmetric tree over time (see Section III-A). However, even this is usually insufficient given the lack of time and the size of the state space. Instead, long-term planning may best be achieved through a different route.

One of the possible options is to create macro-actions, a pre-defined set of consecutive low-level actions that define a single high-level move. Then, MCTS simulations operating in the high-level macro-action space can sample states far ahead in a more diverse manner and hence obtain improved long-term planning. A clear example of this methodology can be found in Real Time Strategy (RTS) games, where groups of simple moves (i.e., transit from A to B, attack a concrete unit) can be included in higher level actions (such as patrol an area or develop a new technology).

However, the PTSP offers an interesting way to balance long-term and short-term planning. Since long-term planning is concerned with the order in which the waypoints are visited and short-term planning is concerned with the actual navigation of the ship, it is possible to tackle these (interdependent) issues separately. The fact that the game is deterministic and single-player allows for this type of long-term planning: the uncertainty due to an opponent's actions or the non-deterministic nature of a game would limit an algorithm's ability to form reliable long-term plans.

In this paper we analyse the performance of MCTS on the PTSP with different combinations of long-term and short-term planning. In particular, we experiment with different routes that take the nature of the game into account in different ways, allowing for different short-term planning patterns to emerge that lead to significant differences in performance.

III. MCTS for Single Player Games

A. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) uses selective sampling to estimate the value of a state: starting from the current state (root), MCTS repeatedly simulates sequences of moves, chosen uniformly at random (default policy), until the end of the game. The value of each terminal state is subsequently back-propagated to gather statistics that allow MCTS to select greedily amongst the states close to the root (tree policy). The selection of nodes for which statistics exist must balance exploration and exploitation in the face of uncertainty and the most common choice is UCB1 for trees [8]:

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

The first term of this equation represents the expected value of applying the action a_i at the state s_t, while the second term favours exploration, considering how often this action is selected from the given node ($N(s_t, a_i)$) in relation to how many times this node has been explored ($N(s_t)$). The constant K is used to weigh these two terms, producing a balance between exploitation and exploration.

The tree created by MCTS is usually asymmetric and new nodes are added to the leaf nodes one at a time. This makes MCTS a promising candidate for real-time games where the time to compute a move is severely limited. An exhaustive description of MCTS and its variations can be found at [2].

B. MCTS for Single-player Real-time Games

MCTS is closely associated with Computer Go where it has led to a breakthrough in performance. Since then, however, MCTS has been applied to a wide range of games, including single-player games (puzzles) and video games. Single-player games are quite different to multi-player games as they not only lack an adversary but also may result in a score rather than a win/loss scenario. MCTS, like minimax, is a technique used in 2-player games and modifications are required to adopt it to the single-player domain. In this section we review studies that applied MCTS to single-player games, real-time games and single-player real-time games. A complete collection of applications of MCTS and its variants to games can be found at [2].

1) Single-Player Games: The Single Player MCTS (SP-MCTS) algorithm was proposed by Schadd et al. [17] and modifies the UCT tree selection to consider the effects of not having an opponent to play against. The authors presented the algorithm as a solution to SameGame, a puzzle played in a 15×15 board where the player has to remove groups of tiles of the same colour. The objective is to clear the board in the least number of moves possible. The authors highlight the difference between single and multiple player games, focusing on the range of values of the score functions. They also propose a meta-search algorithm that resets the tree search with a different random seed, in order to avoid being caught in local minima during the search. The results obtained with SP-MCTS suggest that this algorithm is a good alternative for single player games.

Other authors have also addressed SameGame, such as Cazenave [3], who used Nested Monte Carlo search to achieve good results. Matsumoto et al. [10] applied SP-MCTS to this game, including knowledge information to bias the roll-outs, obtaining better performance. Finally Edelkamp [4] applied an enhanced UCT to beat the highest score registered at the moment.

Morpion Solitaire is another good example of the application of MCTS to single player games: the game consists of repeatedly drawing straight lines on a board to link nodes until no more moves can be made. There are two variations of the problem: the touching and the non-touching versions, in which sharing nodes at the end of the lines is legal or non-legal respectively. Edelkamp et al. [4] obtained good results applying UCT to this problem, and Rosin et al. [14] achieved a new world record, for the 5-line touching version of the problem, applying a modified version of MCTS.

2) Real-Time Games: MCTS has been applied to the 2-player video game Tron where players have to force the opponent to collide against the walls created by their own movements. Samothrakis et al. [15] proposed a standard implementation of MCTS that included heuristic knowledge to prevent the player from self-destructing, and also compared different MCTS variants. The authors conclude that UCT by itself works reasonably well, although some bias needs to be introduced in order to get high quality MC simulations and hence obtain better performance. In a more recent paper, by Den Teuling [18], an enhanced UCT algorithm is applied to the game of Tron. The enhancement, MCTS-Solver, handles simultaneous moves and predicts the outcome of the game without completing a whole simulation, deriving an improvement on the performance of the player.

Another relevant video game is Ms Pac-Man: Samothrakis et al. [16] used MCTS with a 5-player max^n game tree that controls each one of the ghosts separately, showing that MCTS can be used successfully to create strong players for this game. Ikehata and Ito. [5], [6] propose a variant of MCTS that avoids hazardous moves and identifies the most dangerous positions in the maze.

Finally, another category of video games to which MCTS has been applied are real-time strategy games (RTS). Similar to Ms Pac-Man, terminal states are difficult to reach using MC simulations, especially under the tight time constraints, yet modifications may still allow the successful application of MCTS to these types of games. R. Balla and A. Fern [1] apply MCTS to the RTS Wargus. This game focuses on tactical assault planning and the authors achieved better results than human players in 12 maps by assigning abstract actions to groups of units in the game. Similarly, the game ORTS has been tackled by Naveed et al. [11]. The authors used MCTS and Rapidly exploring Random Trees (RRTs) to find paths in the game, showing that MCTS is able to obtain good solutions with less effort, although RRTs still produced stronger players.

3) Single-Player Real-Time Games: Zhongjie et al. [19] have recently applied UCT to the well known game Tetris, a game that shares the single-player and real-time features with the PTSP. In their work, the authors sample the different possible scenarios according to the state of the board and the available pieces. They improved the quality of the simulations by including a pruning mechanism that avoids actions that create holes in the game board, penalising them also in the score value function. Although the number of rollouts per cycle dropped by half because of the complexity of the pruning procedure, the number of pieces correctly placed was increased significantly.

Finally, MCTS was applied previously to an older version of the PTSP [13]. The object of the study was to analyse how the appropriate heuristics can help the tree selection and default policies in order to increase the performance of the algorithm. The results showed that biasing the action selection with domain knowledge improved the solutions obtained, specially when the time allowed to run simulations is very reduced (10ms). Furthermore, the experimentation indicated that MCTS found solutions in a greedy manner, leaving the balance between short and long-term planning as an open problem.

IV. The Physical Travelling Salesman Problem

A. Problem Description

The Physical Travelling Salesman Problem (PTSP) is a modification of the well known combinatorial optimisation problem the Travelling Salesman Problem (TSP). The objective of the TSP is to find the shortest (or quickest) route that will visit each city exactly once, returning to the origin at the end [7]. The PTSP converts the TSP into a single player game where the objective is to drive a ship through a maze in order to visit a determined number of waypoints as quickly as possible.

The actions to navigate the ship are divided into two types: acceleration and rotation. The former can thrust or be idle while the latter corresponds to a fixed rotation to the left, the right or no rotation at all. This leads to a total of six unique actions that can be applied at any one time. The ship may then be navigated to visit each of the waypoints as quickly as possible. To limit the overall length of the game, the ship is required to reach the next waypoint within a pre-defined time limit. The game ends when the time runs out or all waypoints have been visited.

All actions can be seen as forces that modify the position, velocity and orientation of the ship. The ship keeps momentum (or inertia) while friction reduces the speed of the ship over time, stopping it eventually if no thrusting actions are supplied. The physics of the game have an important effect on the problem because it requires an ordering of waypoints that may not correspond to the optimal Euclidean distance based TSP route (see Section V-E).

The PTSP features in the PTSP competition[1] held twice a year at major international conferences. An exhaustive description of the PTSP game, framework and competition is given by Perez et al [12]. Figure 1 shows one of the maps distributed with the competition framework.

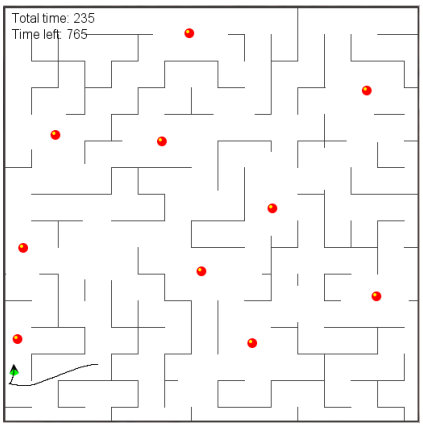

Fig. 1: PTSP map where the ship, at the bottom left, has to visit all waypoints (red dots) scattered around the maze.

Despite its simplicity, the PTSP retains some of the most important aspects of modern video-games: it requires the player to deal with navigation, obstacle avoidance and pathfinding. Furthermore, as it is a real-time game, it does not allow much time to decide what is the next move to make (as would happen in other games such as Chess or Go) or to plan an overall strategy for the game. These characteristics make the PTSP a very good benchmark to try a wide range of techniques that could be exported to real video-games in the future.

B. PTSP Framework: Maps & Controllers

The experiments and results detailed in this paper were obtained using the software and maps from the current PTSP competition (WCCI 2012 Competition), using the following specifications:

- Next waypoint must be reached within 1000 time steps.
- The game advances every 40ms (25 frames per second).
- All maps have 10 waypoints.
- Collisions with obstacles do not damage the ship.

The PTSP competition provides the participants with an online server which has different sets of maps on which to test their controllers. The maps used in this study correspond to the maps of phase 1. The code distributed with the PTSP competition includes a pathfinding library that is used by some of the algorithms presented in this paper. This library creates a grid graph on the navigable sections of the map and may be used to obtain the shortest path between the position of the ship and any other point in the map, accounting for obstacles in the way. Finally, the software also includes several sample controllers that are used as baseline benchmarks in this paper. These controllers are as follows:

- RandomController: executes a random action at every time step (from the six actions available).
- LineOfSight: if there is no unvisited waypoint in the line of sight from the ship's point of view, the controller

[1] www.ptsp-game.net

executes a random action. Otherwise, the ship moves in a straight line towards the visible waypoint.
- GREEDYCONTROLLER: this controller makes use of the pathfinding library included in the framework. It calculates the shortest path from the ship to the closest waypoint and follows it to collect it.

V. EXPERIMENTAL STUDY

A. MCTS for the PTSP

Standard MCTS cannot be applied directly to the PTSP: the open-endedness of the game would prevent any of the MC simulations from reaching a terminal state. It is thus necessary to limit the number of moves (depth) of the default policy. This, however, poses another issue: although one always knows, at any stage of the game, the number of waypoints that were visited up to that stage (and hence one has a perceived notion of quality), this value is typically not fine-grained enough to allow MCTS to distinguish between different rollouts. Many rollouts will not manage to reach an additional waypoint and hence will result in identical scores, making the tree search somewhat akin to the needle-in-a-haystack problem. This issue may be addressed using a more fine-grained value function that assigns distinguishable values to the states encountered by the MC simulations.

The second obstacle that needs to be overcome is the real-time aspect of the game: the number of MC simulations needs to be reduced in order to avoid the algorithm running out of time. It is thus necessary to find the perfect balance between the length of the roll-outs (i.e., the number of moves per MC simulation) and the number of rollouts achievable. If the rollouts are too short it is not possible to obtain enough information to connect a subset of the waypoints. On the other hand, if the MC simulations are too long, only a few roll-outs can be performed at each time step and the uncertainty associated with the action to take is too significant. It is possible to reduce this variance by using domain-specific knowledge to bias the rollouts, albeit at an additional speed reduction.

A central issue is thus the amount of information that can be extracted from the rollouts. There are two main approaches that can be taken. On one hand, it is possible to concentrate the MC simulations on the navigation part in order to get a very fast driver, making use of an ordering of cities obtained *a priori* by a higher level solver. On the other hand, heuristic information can be included in the roll-outs to produce longer and more useful simulations that entail both short-term and long-term planning. The work performed in this paper concentrates on the first approach, making use of the 1 second initialisation time the controller is given: the objective is to obtain a very good driver (short-term look ahead) that is directed by a long-term planner (TSP solver) which has to indicate the order of waypoints to follow.

Since the route connecting all waypoints has to be calculated only once, the remaining problem is reduced in complexity. However, this reduction in complexity comes at the cost of accuracy: since the route and driving are interdependent, any approach that tackles them independently might be sub-optimal. However, it is possible to take the physics of the game (and the ability of the algorithm) into account in various ways when searching for the order of waypoints. Numerous different approaches are investigated in the remainder of this paper.

Furthermore, there are two more parameters of the MCTS algorithm that will be tested: keeping or discarding the tree and saving the best route. The former one, keeping the tree instead of discarding it, is used in order to avoid losing the statistics gathered in previous execution steps. This process is performed by selecting the appropriate child of the root node (depending on the action taken in the last step) and making it the new root. Then, the rest of the tree is discarded (with the exception of the sub-tree under the new root) and a new MCTS iteration starts. Nevertheless, in many games, the whole tree is discarded at every execution step, as the tree can be too large due to a high branching factor.

The latter parameter consists of saving the best route found during the simulations. As the PTSP is a deterministic single-player game, it is possible to keep the best route found and to repeatedly return the actions corresponding to said route until a better one has been found. In two (or more) player games, this is not so useful as the uncertainty of the moves of the opponent must be taken into account. We thus compare here two different move selectors: apply the action of the best route found during the MC simulations or take the action that leads to the best estimated value (standard approach).

B. The Baseline Driver

First an MCTS baseline driver is established that is able to finish the game in most cases. This driver is subsequently used to analyse the differences between short and long-term planning. All experiments have been performed on a set of 20 maps with 5 trials each (i.e., 100 games in total) in order to get some meaningful statistics. The experiments are executed on the competition server under the competition's formal rules, allowing a comparison of the results with those from other participants.

We use MCTS with the following characteristics:

- Tree policy: UCB1 (equation 1), with $K = 0.025$ (value determined empirically).
- Default policy: uniform random.
- The roll-out depth is set to 100 actions.
- The tree is discarded after every execution step, and the best route is not saved.

It is important to have a reliable value function for non-terminal states; the objective of the algorithm is to `maximize` this value function, which is normalised between 0 and 1. Equation 2 is used:

$$s = w + d + c + t \tag{2}$$

where each one of the terms corresponds to the following features of the game state:

- $w = w_n * f$, where w_n is the number of waypoints visited and f a constant (we use $f = 1000$).

- $d = max(900 - d_e, 0)$, where d_e is the Euclidean distance between the ship's position and the closest waypoint in the map.
- $c = -10c_n$, where c_n is the number of collisions of the ship during the simulation.
- $t = 10000 - T$, where T is the time spent in the game (10000 is the maximum duration of a game: number of waypoints multiplied by the maximum number of steps per waypoint).

This approach always drives towards the nearest waypoint until it has been collected. There is no long-term planning that may allow the algorithm to approach a particular waypoint in such a way that the next waypoint after that may be reached in minimum time.

Given the specifications above, the algorithms managed to visit 6.59 waypoints on average (with a standard error of 0.32), taking 2172 time steps on average (standard error of 76.61). The performance of a controller on the PTSP is primarily evaluated by the number of waypoints visited (i.e., 65.9% in this case). The time taken to do so is taken into account as a secondary criterion. In order to improve the number of waypoints visited, some changes are required to the basic setup.

The first modification regards the calculation of the distance to the closest waypoint (i.e, the term d in equation 2): since there are obstacles in the map, a straight line (Euclidean) distance may not be able to provide a reliable estimate of the true cost of reaching a waypoint (and may cause the ship to get stuck against a wall). In the new configuration, the closest waypoint is still chosen depending on the Euclidean distance, but the distance d_e used in the score function is now the cost of the A* path (using the game's built-in graph). Choosing the closest point with the Euclidean distance is an approximation that has been observed to provide better results for efficiency reasons: as it is a faster calculation, more MC simulations are able to be executed at each step.

The second modification was introduced after analysing the behaviour of the driver. In some cases, the ship got stuck against a wall and none of the MC simulations were able to discover an escape route. Hence an UNSTUCK mechanism was implemented. This mechanism rotates the ship, once it has been in the same position for more than a specific amount of time, so it faces in the direction of the next point in the path leading to the closest waypoint.

These two modifications raise the number of waypoints visited on average to 94.6%: the number of waypoints visited is now 9.46 (with a reduced standard error of 0.18) and an average time taken of 2387 time steps (standard error of 86.38). It is important to highlight that there is a direct correlation between the time spent in the game and the number of waypoints collected: although the second experiment resulted in longer execution times, more waypoints were visited and hence it is impossible to draw conclusions regarding the actual speed of the ship. The only fair comparison, attending only to the time spent, must be done between algorithms that obtain similar amounts of waypoints visited. However, an approximation can be obtained by observing the time spent per waypoint (time spent divided by number of waypoints visited). In this case, these algorithms would obtain respective values of 329 and 252 time steps per waypoint, showing that the algorithm with the modifications seems to be faster.

C. Short-term Planning: Driving without a Route

It is possible to further improve the results of the BASELINE DRIVER by systematically tuning the algorithm's parameters. The goal is to find the best possible MCTS controller given no long-term planning at all. The order of cities is determined entirely by the (greedy) selection of the nearest unvisited waypoint at all times.

The following parameters have been tested:

- Two values for K (see equation 1) are compared: 0.025 and $\sqrt{2}$.
- MC simulation depth: 50, 100 and 200 time steps.
- Discarding or keeping the tree after each execution step.
- Saving and applying the actions that take to the best route found, or taking the actions that lead to the best estimated value.

The results of this comparison are shown in Table I. Interestingly, much shorter rollouts tend to lead to overall better performances. In particular, the variants that use a depth of 50 for the MC simulations (specially the top two rows) collect almost 100% of the waypoints with less than 2000 time steps on average (i.e., shorter rollouts also seem to improve speed). On the contrary, variants that use much deeper roll-outs produce significantly worse results and collect less than 86% of waypoints, requiring in excess of 3000 time steps to do so.

The value of K does not have a big impact on the results and $K = \sqrt{2}$ produces only slightly faster drivers. As this value is used widely in the literature, it will be used in the next experiments.

The remaining two parameters analysed, keeping the tree and following the best route, have a somewhat negative impact on the overall performance of the algorithm. The equivalent of the leading configuration (i.e., row one in Table I) that keeps the tree ranks only 7th. Similarly, following the best route found during the simulations increases the time taken to visit the waypoints.

A possible explanation for this may rely on the fact that the ship overshoots a waypoint because of travelling at a very high speed. For instance, the best route could have a high score value because the simulation ended very close to a waypoint. However, the speed of the ship at that point is probably quite high and if the distance to the waypoint is very small, it is likely that the sequence of moves taken during the roll-out of the best route involves a lot of acceleration. If the speed is too high, the ship might not be able to visit the waypoint when the MC simulations really reach it in next steps. On the contrary, more conservative sequences of actions have more possibilities to change the trajectory of the ship in further steps, and eventually visit the waypoint.

MCTS Parameters				Waypoints		Time Spent	
K value	Roll-out depth	Keep tree	Follow best route	Average	Std error	Average	Std error
0.025	50	No	No	9.99	0.01	1955.9	53.67
$\sqrt{2}$	50	No	No	9.98	0.01	1941.57	54.84
0.025	100	Yes	Yes	9.57	0.13	2737.24	85.93
$\sqrt{2}$	100	No	No	9.52	0.17	2376.81	84.57
0.025	100	No	No	9.46	0.18	2386.76	86.38
0.025	100	No	Yes	9.42	0.18	2606.64	87.72
0.025	50	Yes	No	9.4	0.18	1965.97	57.51
0.025	100	Yes	No	8.93	0.24	2308.33	82.07
0.025	200	Yes	Yes	8.57	0.26	2977.58	88.50
0.025	200	No	Yes	8.46	0.27	2925.74	101.52
0.025	50	Yes	Yes	8.2	0.27	3282.57	95.99
0.025	200	Yes	No	8.06	0.28	3024.54	104.98
0.025	200	No	No	8.05	0.27	3049.67	98.63
0.025	50	No	Yes	7.92	0.32	3112.79	95.51
$\sqrt{2}$	200	No	No	7.64	0.29	2770.56	90.71

TABLE I: MCTS, Short-term planning, order by average of waypoints visited.

D. Long-term Planning: Pre-computing the Route

The second part of this empirical study focuses on approaches that make use of pre-computed routes: instead of aiming at the nearest waypoint at the time, the controller now makes use of an *a priori* ordering of cities. This may be established during the initialisation time of the controller before the game starts. This problem essentially corresponds to the classical travelling salesman problem although, as will be shown later, it is possible to take the physics of the game into account to obtain routes that are somewhat longer but more efficient to drive. The method chosen to solve the TSP is the Branch and Bound (B&B) algorithm [9] which is able to provide the optimal solution for 10 cities within the time given. Two variations of B&B are used, depending on how the distances (cost) between the waypoints are computed:

- B&B with Euclidean cost: the cost of going from waypoint A to waypoint B is the Euclidean distance. This algorithm does not take into account the presence of obstacles and is thus expected to be sub-optimal in many cases.
- B&B with A* path cost: using the pathfinding library of the framework, the path between every pair of waypoints is calculated. The travelling cost from one waypoint to another is the length of the path that joins them. This solution resembles an optimal TSP solution.

The long-term planning feature of the algorithm provides an order of waypoints that the ship has to follow. The low-level navigation to do so is handled by the most promising MCTS algorithm from the previous section: a roll-out depth of 50, discarding the tree each time step, not saving the best route found so far and a value of $K = \sqrt{2}$. The value function (i.e., equation 2) is modified to take into account the distance to the next unvisited waypoint in the pre-computed route (rather than the nearest waypoint). Furthermore, a penalty is imposed when the ship accidentally visits a waypoint that is not meant to be collected at that time, in order to follow strictly the established order of cities.

The focus of this set of experiments is to establish the efficiency with which the controller is able to collect all the waypoints and the results are shown in Table II. A first analysis of the results shows that by including any reasonable order of cities (even if sub-optimal), the time spent to solve the problem is reduced significantly (from 1941.57 ± 54.84 to 1830.25 ± 54.25). These results suggest that the addition of long-term planning improves the overall performance significantly. Furthermore, it is evident that the routes computed with A* are more efficient than those computed using Euclidean distance, since the latter does not take the presence of obstacles into account.

	Waypoints		Time Spent	
Algorithm	Average	Std Error	Average	Std Error
Short-term planning	9.99	0.01	1941.57	54.84
B&B (Euclidean)	9.4	0.22	1830.25	54.25
B&B (A*)	9.84	0.10	1744.47	39.29

TABLE II: Results with different TSP solvers.

E. Long Term Planning: Change of Directions

The previous experiment ignored the inter-dependency between the long-term and short-term planning components of the PTSP: an optimal TSP route is not necessarily an optimal PTSP route [13]. In order to address this issue, a different algorithm has been implemented where the cost associated with travelling from one waypoint to another takes into account the changes of direction required in the ship's trajectory. This approach is based on the premise that the ship can visit a series of waypoints connected in a straight (or almost straight) line much more quickly than a series of waypoints that require many changes in direction (and hence loss of momentum).

The most accurate way of computing the optimal ordering of waypoints would be to actually drive the different routes (or sub-routes) using the same controller that is used when playing the game: the base MCTS driver. This, however, would be prohibitively costly given the short initialisation time the controller has to compute the route in the first place (it would essentially solve the problem offline). Instead, we approximate the physics (and driving styles) of the game by

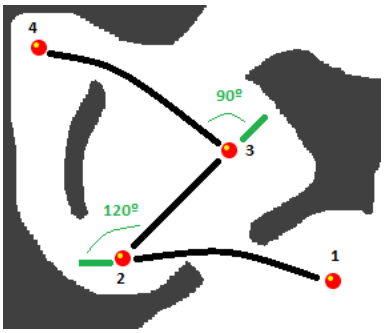

Fig. 2: B&B using costs as lengths of the shortest path.

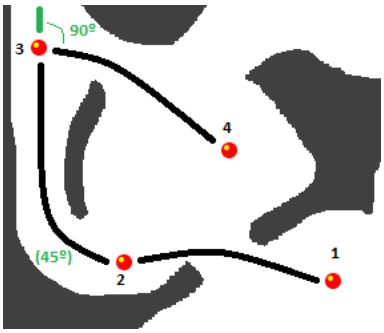

Fig. 3: B&B adding changes of direction to costs calculation.

taking the angles into account, producing a route that is likely to be sub-optimal yet superior to routes obtained using the classical TSP approach.

Figure 2 depicts a portion of a map that shows the path that a (classical) TSP solver would suggest if only distances between nodes would be taken into account. This route requires two significant changes of direction (120 and 90 degrees respectively). A PTSP-optimised route is shown in Figure 3: this route takes into account the changes of direction and reduces the number of sharp turns (a single sharp turn of just above 90 degrees is now required).

To obtain PTSP-optimised routes, it is necessary to account for the number of turns in a route by introducing a cost factor when searching the optimal TSP route. The final cost of a path is thus obtained by multiplying the normal length of the A* path by a factor that depends on the summation of angles from the route. To analyse if there is a significant difference after including the changes of direction in the path cost, Figure 4 shows a base case. In this test map, it is clear that the order of the waypoints affects the quality of the solution significantly.

The experiment from the previous section is repeated using the new TSP solver that takes the angles of the route into account. It is evident that the performance is slightly better, as the results have been improved, 1703.07 ± 40.12 against 1744.47 ± 39.29, although the difference is not especially big. The reason for the only small rate of improvement is that the new TSP solver produced routes that differ in only 9 of the 20 maps. In other words, even accounting for the angles in the route, always driving towards the nearest waypoint results in the same overall order of waypoints in 11 of the 20 maps. If the analysis is concentrated only on those nine maps, the improvement is more noticeable: a run that executes 15 times on each of these nine maps shows that the TSP solver that considers angles outperforms the standard TSP solver, with an average time taken of 1753.98 ± 39.25 compared to 1820.76 ± 37.81; it obtains better results in seven out of nine maps.

A final evaluation of comparison includes the best controllers submitted to the PTSP competition: all controllers are evaluated 5 times per map to obtain averages of the number of waypoints and time spent, discarding the two worst games on each map. The controllers are then ranked for each map by the number of waypoints visited and the time required to do so. Points are awarded for each rank as follows: 10 points for the best controller, 8 for the second, 6 for the third, 5 for the fourth and so on until 1 point for the eighth. The results of this comparison are shown in Table III. The results indicate that the algorithms presented here are competitive, as they got better results than other approaches during the early stages of the competition. Nevertheless, it is important to mention that this does not imply that these algorithms are the best for this problem, as the other participants improved their controllers during the competition, obtaining better performances. This, however, is not the ultimate goal since the focus of this empirical study lies on the relative differences in performance obtained using different variants and combinations of short-term and long-term planning. Nevertheless, it is reassuring to see that the results are good enough to place these algorithms at the top of the rankings. This also demonstrates that taking into account the changes of direction when computing the order of waypoints produces excellent results.

VI. Conclusions

This paper demonstrates the importance of long-term planning when using Monte Carlo Tree Search (MCTS) for real-time and open-ended (video) games, as exemplified on the Physical Travelling Salesman Problem (PTSP). The real-time element of these games, where the time to compute the next move is severely limited, prevents sample-based algorithms such as MCTS from looking far enough ahead to behave optimally. This issue may be addressed by solving the long-term and short-term aspects of the game separately: the solution proposed in this paper shows how providing the order of cities (i.e., solving the long-term planning aspect) makes a noticeable difference in the performance of the algorithm, reducing the time spent solving the problem and still proving a high rate of success in visiting all the waypoints of the maps.

There is plenty of room for improvement: taking into account not only the changes of direction, but also the way the driver actually moves through the maze, could lead to better results. In particular, the speed at which the ship is supposed to arrive at each waypoint has not been considered in this study, and that could make a big difference. Likewise, it would be interesting to see whether MCTS could find the

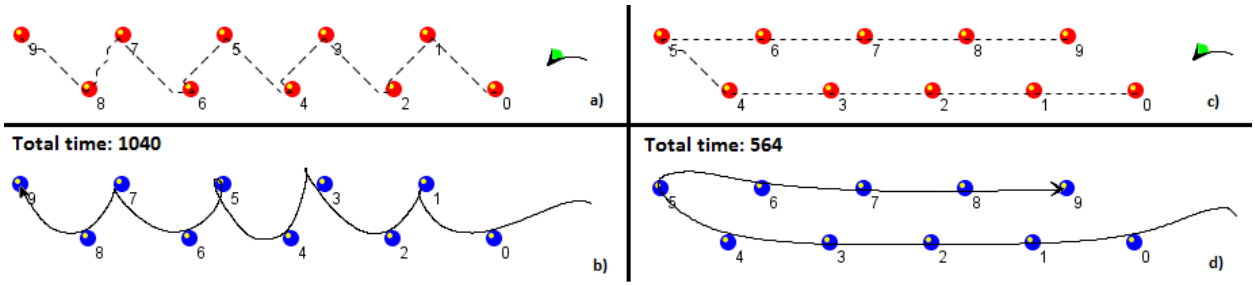

Fig. 4: This figure shows the results of following different TSP routes. The top row depicts the order of cities to follow, whilst the bottom row shows an actual game played by the controllers. The left column (figures a and b) presents the problem solved by a TSP that only takes into account the distances between the waypoints. The right column, figures c and d, shows the performance of the controller that includes the changes of direction in the route calculation. As can be seen, the latter algorithm is able to solve the problem in 564 steps, while the former only can do it in 1040.

Controller \Map:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Total**
TSP ANGLES	8	**10**	**10**	**10**	**10**	8	**10**	8	**10**	**10**	8	**10**	**10**	**10**	**10**	8	6	8	8	**10**	**182**
TSP NORMAL	**10**	8	8	8	8	**10**	6	**10**	8	8	**10**	8	8	8	8	6	**10**	**10**	**10**	8	170
ST3F1	6	5	6	6	5	6	8	6	6	6	4	6	6	5	4	5	5	5	6	6	112
PUROFVIO	4	6	5	5	6	4	4	5	4	4	5	5	5	6	6	10	8	6	5	4	107
PHILSTER	5	4	4	4	4	5	5	4	5	5	6	4	4	4	5	4	4	4	4	5	89
GREEDY	3	3	3	3	3	3	3	3	3	2	3	3	3	3	3	3	3	3	2	2	57
LINEOFSIGHT	2	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	3	3	43
RANDOM	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	20

TABLE III: This figure presents the rankings of some controllers following the point award scheme of the PTSP competition. It shows eight controllers (rows of the table) evaluated in 20 maps (columns), during the first phase of the WCCI PTSP competition, which corresponds to controllers submitted from 1st March to 1st April 2012. The first two controllers are the ones shown in this paper: the TSP solver that considers direction changes (TSP ANGLES) and the TSP solver that does not (TSP NORMAL). Then, the three best participants' controllers (ST3F1, PUROFVIO and PHILSTER) and finally the three sample controllers distributed with the software (GREEDY, LINEOFSIGHT and RANDOM controllers), described in IV-B.

optimal racing line given the distribution of waypoints and obstacles. These issues will be investigated in future research.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Radha-Krishna Balla and Alan Fern. UCT for Tactical Assault Planning in Real-Time Strategy Games. In *Proc. 21st Int. Joint Conf. Artif. Intell.*, pages 40–45, Pasadena, California, 2009.

[2] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:1:1–43, 2012.

[3] Tristan Cazenave. Nested Monte Carlo Search. In *Proc. 21st Int. Joint Conf. Artif. Intell.*, pages 456–461, Pasadena, California, 2009.

[4] Stefan Edelkamp, Peter Kissmann, Damian Sulewski, and Hartmut Messerschmidt. Finding the Needle in the Haystack with Heuristically Guided Swarm Tree Search. In *Multikonf. Wirtschaftsinform.*, pages 2295–2308, Gottingen, Germany, 2010.

[5] N. Ikehata and T. Ito. Monte Carlo Tree Search in Ms Pac-Man. In *Proc. 15th Game Progr. Workshop*, pages 1–8, 2010.

[6] N. Ikehata and T. Ito. Monte Carlo Tree Search in Ms Pac-Man. In *Proc. IEEE Conf. Comput. Intell. Games*, pages 39–46, 2011.

[7] D. S. Johnson and L. A. McGeoch. *The Traveling Salesman Problem: A Case Study in Local Optimization*. John Wiley and Sons, Ltd., 1997.

[8] Levente Kocsis and C. Szepesvári. Bandit based Monte Carlo planning. *Machine Learning: ECML 2006*, 4212:282–293, 2006.

[9] A. Land and A. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[10] Shimpei Matsumoto, Noriaki Hirosue, Kyohei Itonaga, Kazuma Yokoo, and Hisatomo Futahashi. Evaluation of Simulation Strategy on Single-Player Monte Carlo Tree Search and its Discussion for a Practical Scheduling Problem. In *Proc. Int. Multi Conf. Eng. Comput. Scientists*, volume 3, pages 2086–2091, Hong Kong, 2010.

[11] M. Naveed, D. Kitchin, and A. Crampton. Monte Carlo Planning for Pathfinding in Real-Time Strategy Games. In *Proc. 28th Workshop UK Spec. Inter. Group Plan. Sched.*, pages 125–132, 2010.

[12] D. Perez, P. Rohlfshagen, and S. Lucas. The Physical Travelling Salesman Problem: WCCI 2012 Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2012.

[13] Diego Perez, Philipp Rohlfshagen, and Simon Lucas. Monte carlo tree search for the physical travelling salesman problem. In *Proceedings of EvoApplications*, volume 7248, pages 255–264, 2012.

[14] Christopher D. Rosin. Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In *Proc. 22nd Int. Joint Conf. Artif. Intell.*, pages 649–654, Barcelona, Spain, 2011.

[15] Spyridon Samothrakis, David Robles, and Simon M. Lucas. A UCT Agent for Tron: Initial Investigations. In *Proc. IEEE Symp. Comput. Intell. Games*, pages 365–371, Dublin, Ireland, 2010.

[16] Spyridon Samothrakis, David Robles, and Simon M. Lucas. Fast Approximate Max-n Monte Carlo Tree Search for Ms Pac-Man. *IEEE Trans. Comp. Intell. AI Games*, 3(2):142–154, 2011.

[17] Maarten Peter Dirk Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume Maurice Jean-Bernard Chaslot, and Jos W. H. M. Uiterwijk. Single-Player Monte Carlo Tree Search. In *Proceedings of Computer Games, LNCS 5131*, pages 1–12, 2008.

[18] N. Den Teuling. Monte carlo tree search for the simultaneous move game tron. Technical report, Univ. Maastricht, Netherlands, 2011.

[19] Cai Zhongjie, Dapeng Zhang, and Bernhard Nebel. Playing tetris using bandit-based monte carlo planning. In *Proceedings of AISB 2011 Symposium: AI and Games*, 2011.