Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions

Diego Perez   Edward J. Powley   Daniel Whitehouse
Philipp Rohlfshagen   Spyridon Samothrakis   Peter I. Cowling
Simon M. Lucas

Abstract

This paper presents a number of approaches for solving a real-time game consisting of a ship that must visit a number of waypoints scattered around a two-dimensional maze full of obstacles. The game, the Physical Travelling Salesman Problem (PTSP), which featured in two IEEE conference competitions during 2012, provides a good balance between long-term planning (finding the optimal sequence of waypoints to visit), and short-term planning (driving the ship in the maze). This paper focuses on the algorithm that won both PTSP Competitions: it takes advantage of the physics of the game to calculate the optimal order of waypoints, and it employs Monte Carlo Tree Search (MCTS) to drive the ship. The algorithm uses repetitions of actions (macro-actions) to reduce the search space for navigation. Variations of this algorithm are presented and analysed, in order to understand the strength of each one of its constituents and to comprehend what makes such an approach the best controller found so far for the PTSP.

1 Introduction

Artificial Intelligence (AI) for real-time video games has become an important field of study in recent years, and within the field there are a diverse range of interesting challenges and benchmarks. The games studied vary greatly in complexity and sometimes several problems have to be tackled to progress within small decision time budgets.

In this paper, the focus is on the Physical Travelling Salesman Problem (PTSP), a single-player real-time video game where the player needs to complete a puzzle providing actions at a rate of one every 40ms. This game featured in two international IEEE competitions during 2012 [1], receiving numerous entries that tried to solve the problem employing a range of different techniques.

This paper is centred on the application of Monte Carlo Tree Search (MCTS) [2] to the PTSP, using different planners and mechanisms to reduce the search space and to improve performance. MCTS is based on two main concepts: the first one is the balance between exploration and exploitation of the search space, using a tree selection mechanism based on the Upper Confidence Bound for Trees (UCT) [3]. The second concept is the use of Monte Carlo (MC) simulations to compute estimated rewards for applying different actions in certain states. The controller explained here is the winner of both editions of the competition. The goal of the paper is to provide insight into the various components of the controller, analysing its strengths, weaknesses and the importance of each one of its constituents.

MCTS is a game tree search method that has shown outstanding performance where some other techniques, such as minimax or depth first search, have failed. A clear example of this is the game of Go, where MCTS is the first algorithm able to provide a professional level of play in some versions of the game [4]. Since then, MCTS has been applied to a wide range of different games and applications. Browne et al. [5] present, in a complete survey, the description of the algorithm, variations and applications.

In the domain of real-time games, MCTS has been applied to different problems. For instance, Samothrakis et al. [6] compared different MCTS variants in the game of Tron, including heuristics that dissuaded the player from performing suicidal movements. Also, Den Teuling [7], in a more recent paper, proposes a UCT player that handles simultaneous moves and predicts outcomes without completing a full simulation.

Another domain where MCTS has been successfully applied is Ms Pac-Man. In this game, the player controls Ms Pac-Man aiming to eat all the pills scattered in the maze while avoiding being captured by the ghosts. Ikehata et al. [8] propose a version of MCTS that identifies dangerous moves and zones in the map where it is likely to be captured.

Single-player real-time games, such as the PTSP, have also been used as benchmarks for MCTS techniques. For instance, Zhongjie et al. [9] applied MCTS to the well known game of Tetris. The authors included a novel mechanism in MCTS that pruned branches of the tree to remove those actions that would create holes in the array of blocks. The modification increased the time needed to decide a move, but the actions taken produced better results than the unmodified version of the algorithm.

The PTSP itself has also been approached using MCTS techniques. A previous version of the problem, where the maps had neither obstacles nor boundaries, was used by Perez et al. [10] to suggest the importance of heuristic knowledge in the tree search policy and the Monte Carlo simulations. Finally, the algorithm described in this paper won the PTSP competition twice in 2012, making use of MCTS. The first version of the winning controller is explained in [11]. One of the key aspects of this controller is the use of *macro-actions*. As the PTSP is a single player and deterministic game, it is very convenient to group single actions into pre-defined sequences to reduce the search space size. For instance, a macro-action would be the repetition of a single action a for T time steps. These abstractions can help to reduce the branching factor and the MC simulations of the algorithm can look further ahead with perfect accuracy. The idea of macro-actions (or temporally extended actions) has a long history in Artificial Intelligence (see [12]) and has been used extensively in domains where the size of the state space would make the cost of searching prohibitive with current computational resources(e.g. [13]).

Balla and Fern [14] applied UCT to the Real Time Strategy game Wargus. This game is based on tactical assault planning, where multiple units are concurrently performing actions in the game and different actions, executed simultaneously, can take a variable duration. The authors discretize this continuous state in an abstraction that simplifies the characteristics of the current game layout: transitions between states are managed by actions that affect a group of units, reducing the size of the tree search and allowing more efficient simulations.

Abstracted action spaces have also been used, for instance, by Childs et al. [15] (and recently revisited by Van Eyck et al. [16]), who applied combined groups of actions when using UCT in the artificial game tree P-Game. A P-Game tree is a minimax tree where the winner is decided by a counting mechanism on the final state of the board. In their study, the authors group several similar actions in a single move to reduce the branching factor, obtaining promising results. Powley et al. [17] employed a similar technique when applying MCTS to the card game of Dou Di Zhu, where a single move choice is sometimes split into two separate and consecutive decisions to reduce branching factor at the expense of tree depth.

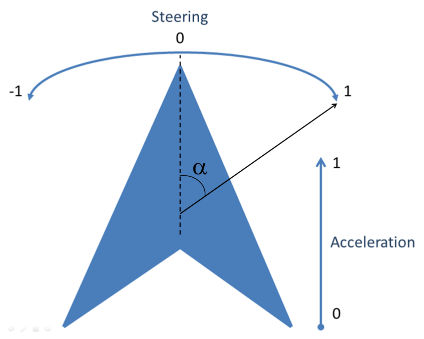
The rest of this paper is organised as follows: Section 2 describes the PTSP game and Section 3 explains the general architecture of the PTSP controller. Section 4 introduces then the tree search methods employed in this research, and Sections 5, 6 and 7 describe the different components and parameters of the controller: macro-actions, value functions and TSP solvers, respectively. The experimental setup is described in Section 8 and the results and analysis are presented in Section 9. Finally, the conclusions and future work are presented in Section 10.

2 Physical Travelling Salesman Problem

2.1 The Game

The Physical Travelling Salesman Problem (PTSP) is an adaptation of the Travelling Salesman Problem (TSP) to convert it into a single-player real-time game. The TSP is a well known combinatorial optimisation problem where a series of cities (or nodes in a graph) and the costs of travelling between them are known. A salesman must visit all these cities exactly once and go back to the starting city following the path of minimum cost.

In the PTSP, the player (i.e. the salesman) governs a spaceship that must visit a series of waypoints scattered around a maze as quickly as possible. The PTSP is a real-time game, implying that an action must be supplied every 40 milliseconds. The available actions are summarized in Figure 1. These actions are composed of two different inputs applied simultaneously: acceleration and steering. Acceleration can take two possible values (*on* and *off*), while steering can turn the ship to the *left*, *right* or keep it *straight*. This leads to a total of six different actions. In case the controller fails to return an action after the time limit, an *NOP* action (ID: 0) is applied, which performs no steering and no acceleration.



Action ID	Acceleration	Steering
0	No (0)	No (0)
1	No (0)	Left (-1)
2	No (0)	Right (1)
3	Yes (1)	No (0)
4	Yes (1)	Left (-1)
5	Yes (1)	Right (1)

Figure 1: Action space of the PTSP

The state of the ship is kept from one game step to the next, and three different vectors are modified after applying a single action. The orientation of the ship is changed as shown in Equation 1, given the ship's orientation in the last step d_t and the rotation angle α (a fixed angle that can be positive or negative, depending on the sense of the steering input, or 0 if the ship is told to go straight). Equation 2 indicates how the velocity vector is modified, given the previous velocity v_t, the new orientation, an acceleration constant (K) and a frictional loss factor (L). In this case, the acceleration input determines the value of T_t: being 1 if the action implies thrust or 0 otherwise. Finally, Equation 3 updates the position of the ship by adding the velocity vector to its location in the previous game step (p_t). This physics model keeps the inertia of the ship between game cycles, making the task of navigating the ship more challenging.

$$d_{t+1} := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} d_t \tag{1}$$

$$v_{t+1} := (v_t + (d_{t+1} T_t K))L \tag{2}$$

$$p_{t+1} := p_t + v_{t+1} \tag{3}$$

The obstacles of the maze do not damage the ship, but hitting them produces an elastic collision, which modifies the velocity of the ship (both in direction and magnitude). An example of a map distributed with the game is shown in Figure 2.

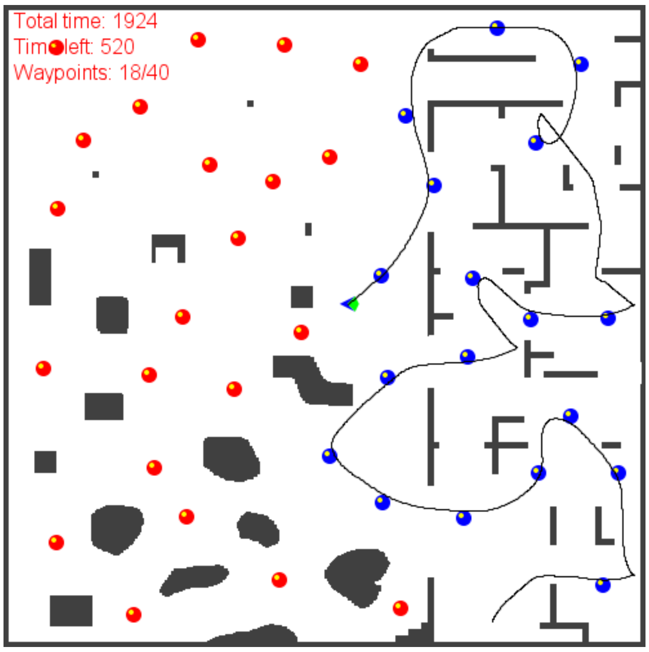


Figure 2: Example of a PTSP map.

In addition to returning an action every 40 milliseconds, a controller designed for the PTSP must respect two other time limits: an initialization time and a limit to reach the next waypoint in the game. Both times depend on the number of waypoints present, which in the experiments described in this paper are 30, 40 and 50. The initialization time, that can be used by the controller to plan the route to follow before the game begins, is set to 1 second per each 10 waypoints. The time to reach the next waypoint depends on the density of the waypoints in the maze: maps with 30 waypoints allow 700 time steps, 40 waypoints imply 550 game ticks and 50 permit the usage of 400

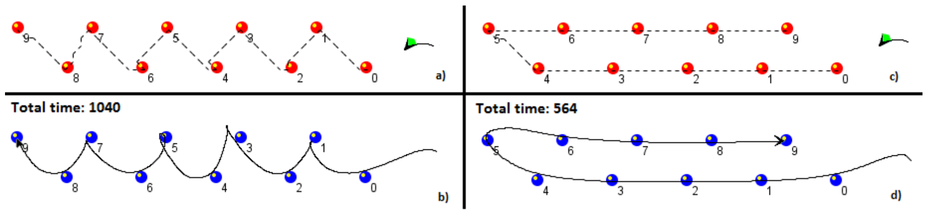


Figure 3: Example of different routes and performances in PTSP. Figures a and c show the order of waypoints according, respectively, to a distance-based TSP solver and a second planner that takes the physics of the game into account. Figures b and d show the trajectory described while following these routes. Note that the distance-based route completes the game in 1040 game ticks, while the physics-based one visits all the waypoints in only 564 steps.

game steps. These counters are set to this limit every time a waypoint is visited, and are decreased by 1 at every cycle. If the value reaches 0, the game is over. There is also a second counter that counts the total number T of steps of the game.

The final score of the game is defined by the tuple (W, T) where W is the number of waypoints visited and T is the total time used. Between two matches, the one that gets more waypoints is considered to be the best. In case of a draw, the one that finished the game in fewer time steps wins.

At every game step, the controller is supplied with a copy of the current game state and point in time the move is due. The game state contains information about the ship itself (position, velocity, orientation), the game (time left, number of waypoints visited), the map (position of obstacles and path-finding queries) and waypoints (positions and whether they have been visited or not). However, the most important feature of this game state is that it provides a *forward model* to run simulations. In other words, it is possible to check what the future states of the game would be if a specific sequence of moves were played. The PTSP is a deterministic and single-player game and the simulations performed are completely accurate. This makes tree search and simulation approaches especially well suited for this kind of problem.

2.2 PTSP versus TSP

A reasonable approach to solving the PTSP is to divide the problem into its two constituents: the order of waypoints to follow and navigating the ship to each one of them. However, these two problems are highly interdependent, as ignoring the physics model of the game may lead to a suboptimal order of waypoints. In particular, if the order of waypoints is obtained only taking the distances between the waypoints into account (calculated, for instance, using the A* algorithm) the final result may be suboptimal. It is worth noting that the final objective is to minimize the time taken to visit all waypoints, and not the distance. Hence, routes that might not seem optimal because of their length could provide better solutions if the ship is able to drive them quickly. In general, it has been suggested [1] that routes that minimize the changes of direction and cover waypoints in straight (or almost straight) lines are better for the PTSP. Figure 3 (from [1]) shows a clear example of these situations in PTSP.

2.3 The PTSP competition

The PTSP featured in two competitions in 2012, held at two international conferences: the IEEE World Congress on Computational Intelligence (WCCI) and the IEEE Conference on Computational Intelligence and Games (CIG).

In both competitions, the participants were able to create controllers in Java by downloading a starter kit that contains some sample controllers and 10 different maps. A server, accessible at `www.ptsp-game.net`, was running continuously, receiving submissions and executing the controllers in these 10 maps and other groups of maps, created to provide a test mechanism for the participants in the machine where the final evaluation would be made.

Following the deadline of the competition, the controllers played a set of 20 new maps that had never been seen or played by the participants before. Each controller was executed 5 times on each map. The final result was the average of the best three matches, featuring waypoints and time taken. The players, ranked according to number of waypoints visited and time taken, were awarded following a Formula One style scoring scheme (described in Section 9.3). The winner of the competition is the player who submitted the controller that achieved the most points across all maps.

In the WCCI competition [18], only maps with 10 waypoints were considered. However, in the CIG edition of the competition, the maps were changed to contain 30, 40 and 50 waypoints. In this study, the maps used for the experiments are the ones employed for the final rankings of the CIG Competition.

The winning method of both bot competitions was the entry by Powley et al., who described their WCCI controller in [11]. This controller, based on MCTS, macro-actions and a physics-based TSP solver, is the base for the experiments described in this paper.

3 The PTSP Controller

The following sections in the paper describe the different constituents of the PTSP controller, such as the TSP solvers, search methods and macro-action discretization. Figure 4 shows the general architecture of the controller, which is common for all configurations tested in this research.

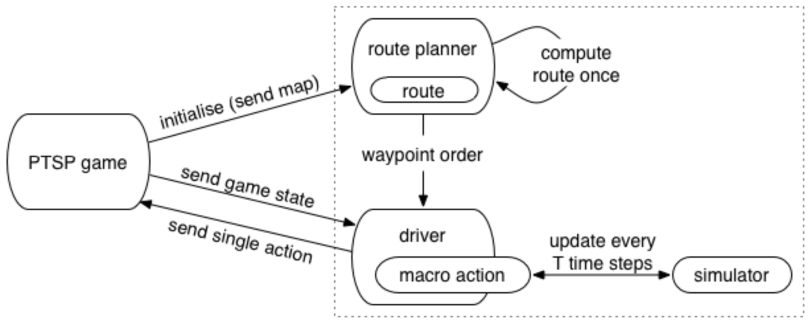


Figure 4: General scheme of the PTSP controller.

During the initialization step, the *TSP Route Planner* (or *TSP Solver*) obtains the order of waypoints to follow during the game. Then, the game requests an action from the *PTSP Controller* (also mentioned here as the *Driver*) at every game step, providing information about the game state, until the game is ended. In each one of these cycles, the controller simulates actions in the game, deciding which will be the next action to execute. As will be explained later in Section 5, macro-actions are used to discretize the search space. While the controller is executing a given macro-action at time t, the simulations are performed in order to find out which is the best macro-action to execute at time $t + T$ (where T is the length of all macro-actions).

4 Search methods for the PTSP controller

Three different search methods have been employed in this study to tackle the PTSP: Depth First Search, Monte Carlo simulations and Monte Carlo Tree Search. All these methods share the same time constraint: a move decision must be made within the time given (40ms). This feature determines the way the algorithms are designed, as adjusting to this time budget makes it impossible to reach the end of the game in the vast majority of cases.

Additionally, the PTSP is an open-ended game: it is not possible to find a state in the search space where all waypoints are visited just by applying uniformly random action selection. Indeed, even reaching the next waypoint to visit is not normally achievable via uniform random simulations. It is therefore essential to add a heuristic value function capable of evaluating the quality of the different states found in the game (for a description of these, see Section 6). This section details the search procedures employed in this study.

4.1 Depth First Search

Depth First Search (DFS) is a traditional graph or tree search algorithm. It searches exhaustively at each node of the tree, iteratively expanding the next unvisited child and searching deep until a non-expandable node is reached. The search navigates up the tree to find the next unexpanded node and continues this process, until all nodes below the root have been visited.

Starting from the root, which represents the current state of the game, actions are applied that lead to future game states, using the *forward model* mentioned in Section 2. These future states, derived after applying an action, are taken as children of the parent node. Finally, the action that led to the best score achieved is taken as the move to play in the game.

In order to fulfil the time budget limitations, a maximum depth for the search needs to be decided. In this case, it has been determined empirically that when depth 4 has been reached, DFS cannot expand any further without exceeding the imposed time limit. Therefore, when a node in this depth is reached, an evaluation is performed in order to assign a score to the state found, using one of the state evaluators described in Section 6. Owing to this depth limitation, the number of states that are evaluated for each decision is $6^4 = 1296$.

4.2 Monte Carlo Simulations and UCB1

Monte Carlo (MC) methods are based on random simulations that sample the search space uniformly or guided by heuristics. In the case of the PTSP, simulations are performed from the current state of the tree (or root), choosing a sequence of actions uniformly at random until a maximum depth has been reached. The state of the game is then evaluated. Different values for the simulation depth have been tried in the experiments described in this paper. The depth of the simulation and the 40ms time budget sets the limit of how many simulations can be performed.

If actions are picked uniformly at random, it is expected that each action is selected one sixth of the time, producing an evenly distributed search. This approach, however, does not benefit from the fact that one or more actions can be better than others, as suggested by the scores obtained by the evaluations at the end of the simulations.

To overcome this problem, a *multi-armed bandit* selection policy is employed. Multi-armed bandits is a problem from probability theory where each one of the multiple slot machines produces consecutive rewards $r_t : r_1, r_2, \dots$ (for time $t = 1, 2, \dots$) driven from an unknown probability distribution. The objective is to minimize the regret, i.e. the losses of not choosing the optimal arm. Multi-armed bandit policies select actions in this problem by balancing the exploration of available actions (pulling the arms) and the exploitation of those that provide better rewards (optimisation in the face of uncertainty).

Auer et al. [19] proposed the *Upper Confidence Bound (UCB1)* policy for bandit selection:

$$a^* = \operatorname*{arg\,max}_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{4}$$

The objective is to find an action a that maximizes the value given by the UCB1 equation. Here, $Q(s,a)$ represents the empirical average of choosing action a at state s. It is calculated as the accumulated reward of choosing action a divided by how many times this action has been picked. $Q(s,a)$ is the *exploitation* term, while the second term (weighted by the constant C) is the *exploration* term. Note that, if the balancing weight $C = 0$, UCB1 follows greedily the action that has provided the best average outcome so far. The exploration term relates to how many times each action a has been selected from the given state s ($N(s,a)$) and the amount of selections taken from the current state ($N(s)$). When action a is chosen, the values of $N(s,a)$ and $N(s)$ increase. The effect of this is that the exploration term for other actions different than a increase, allowing a more diverse exploration of the different available actions in future iterations of the algorithm.

The value of C balances between these two terms. If the rewards $Q(s,a)$ are normalized in the range $[0,1]$, a commonly used value for C in single-player games is $\sqrt{2}$. It is also important to notice that, when $N(s,a) = 0$ for any action, that action must be chosen. In other words, it is not possible to apply UCB1 if all actions from a state have not been picked at least once.

When the time budget is over, the action to execute in the game is the one that has been chosen more times by UCB1.

4.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search technique that creates a tree by performing Monte Carlo simulations. The best known version of MCTS is Upper Confidence Bound for Trees (UCT), proposed by Kocsis and Szepesvári [3, 20]. UCT uses UCB1 (as described in Section 4.2) to build a potentially asymmetric tree that is grown towards the more promising parts of the search space. Every time UCT selects an action that leads to a state which is not represented in the tree, a new node is added to it and a Monte Carlo simulation is started from there.

The MCTS algorithm can be seen as a four-step process that is repeated for several iterations. During the first step (*Selection*), the tree selection policy (e.g. UCB1 for UCT) selects the next action to take. The action leads to a node in the tree. If the node is already present in the tree then the selection policy is applied again (and so on recursively); if not, it is added as a new leaf node (*Expansion* step). An MC simulation is initiated from this node until a pre-determined depth or the end of the game. The state reached at the end of the simulation is evaluated (*Simulation* step). This MC simulation is driven by a default policy, that can be uniformly at random (as it is in this study) or guided by a heuristic.

The last step in the iteration is *Backpropagation*. During this phase, the reward obtained by the evaluation function is back-propagated throughout the visited nodes until the root. All the nodes traversed update their internal statistics: $N(s)$, $N(s, a)$ and $Q(s, a)$, gathering knowledge for the tree selection phase in the next iterations of the algorithm.

As in the previous cases, the depth of the MC simulation must be limited as, if not, it would not be possible to reach the end of the game within the allowed time budget. These depth values are defined in the experimental setup, but it is important to notice that the simulation depth corresponds to the number of actions taken from the root of the tree, and not from the new expanded node at each iteration. This is done in order to provide the same look ahead from the children of the root despite the asymmetric tree growth.

5 Macro-actions in the PTSP

The algorithms presented in this paper treat the PTSP as two different (but related) sub-problems: navigating the ship and planning the order of waypoints to be followed. This section describes the approach used to drive, or steer, the ship along the route chosen by the methods described in Section 7, i.e. how to follow the paths from one waypoint to the next. (In line with games literature, "steering" here includes acceleration as well as rotation of the ship.) An input to the ship is sent once every 40ms, in order to steer the ship around obstacles and towards waypoints.

The steering problem for PTSP can be considered as a sequential decision problem on a tree. It can also be considered as a problem on a directed graph, i.e. allowing multiple paths to the same node and possibly cycles, but for simplicity the graph is assumed to be a tree. The nodes of the tree are *states* and the edges are *actions*. A *macro-action* M is defined as a sequence of actions $M = \langle a_1, \ldots, a_n \rangle$. *Executing a macro-action* corresponds to playing out the sequence of actions contained within the macro-action. A decision tree of macro-actions can be built: the node set of this tree is a subset of the node set of the original decision tree.

In the PTSP, the set of legal actions from a state (i.e. the set of edges out of the corresponding node) is the same for all states. If this was not the case, more care would be needed in defining macro-actions: if the macro-action is to be applied at state s_0, and s_i is the state eventually obtained by applying all actions, then a_{i+1} must be a legal action from s_i.

For the PTSP, the purpose of macro-actions is to reduce the size of the problem and to increase the ability of tree search methods to perform forward planning. This can be achieved by reducing the granularity of possible paths and preventing the ship from making small (sometimes meaningless) adjustments to speed and direction. The macro-actions used in this paper are arguably the simplest possible, consisting of executing one of the six available actions (see Figure 1), for a fixed number of time steps T.

More complex macro-actions were tested, such as rotating to one of several specified angles while thrusting or not thrusting. One problem that arose was that search methods did not perform well when different actions took different lengths of time to execute: since the evaluations used (Section 6) are implicitly functions of distance, a search method optimising these value functions tends simply to favour longer macro-actions over shorter ones. Having each depth of the tree corresponding to the same moment in time and having the search roll out to a fixed depth means instead that search methods will optimise the path length (by maximizing distance travelled in a fixed amount of time).

On a map of 50 waypoints, the game always takes less than 20000 time steps (a longer time would imply that the time limit between waypoints was exceeded at least once). Thus the decision tree has up to 6^{20000} nodes. The game takes less than $\frac{20000}{T}$ macro-actions, therefore the size of the macro-action tree is bounded above by $6^{\frac{20000}{T}}$, which represents a hundreds of orders of magnitude reduction in the size of the problem to be solved (when $T \geq 2$). To illustrate this, let us make a conservative estimate of 2000 time steps for the average length of a game, and set $T = 15$. The game tree contains $6^{2000} \approx 10^{1556}$ states, whereas the macro-action tree contains $6^{\frac{2000}{15}} \approx 10^{103}$ states. The size of the macro-action tree in this example is comparable to the game tree size for a complex board game: for comparison, the number of states in 9×9 Go is bounded above by $81! \approx 10^{120}$. The macro-action tree is of the order 10^{1453} times smaller than the full game tree.

The parameter T controls the trade-off between the granularity of possible paths and the forward planning potential for tree search. Since one rotation step corresponds to a rotation of $3°$, a setting of $T = 30$ restricts the ship to only making $90°$ turns. (Note that the ship may have an initial velocity, and may thrust while turning, so the restriction to $90°$ turns does not restrict the path to $90°$ angles). When using this setting, search algorithms will find paths that have to bounce off walls or follow convoluted routes to line up with waypoints. A choice of $T = 10$ corresponds to $30°$ turns, which allows for a finer control of the ship and smoother paths. The difference is illustrated in Figure 5 where the path with $90°$ turns is more jagged (and takes longer to follow) than the one with $30°$ turns. On the other hand, when $T = 10$ reaching a depth d in the search tree corresponds to a point in time sooner than with $T = 30$. This impacts the steering controller's ability to plan ahead.

It is worthwhile mentioning that a different approach could also be possible: instead of executing all the single actions from the best macro-action found, one could only apply the first one. Macro-actions are employed during MC simulations, but all routes are possible for the driver as the next best macro-action is computed every 40ms. This approach, however, allows less time for deciding the next move, as only one game step

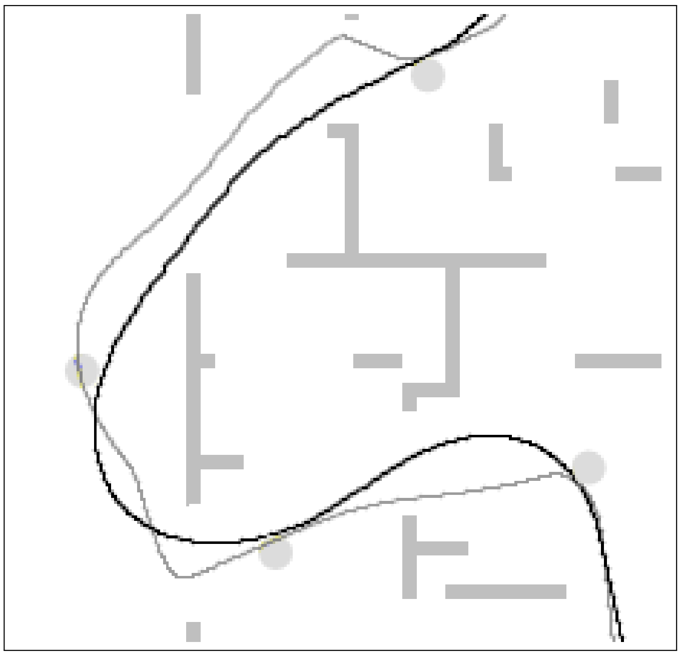


Figure 5: Examples of the path followed by the MCTS controller. The light grey line has $T = 30$ corresponding to $90°$ turns. The black line has $T = 10$ corresponding to $30°$ turns.

can be used to pick the next action to take.

6 Value functions

The goal state in PTSP (a state in which all waypoints have been collected) can be several thousand time steps, or several hundred macro-actions, into the future. Finding such a state purely by tree search is intractable. Instead the search is performed to a fixed depth, applying a heuristic evaluation function to the resulting nonterminal state, and allowing the tree search to optimise this heuristic value.

The value functions are based around the "shortest path" distance to a target waypoint, taking obstacles into account. Section 6.1 describes the precomputed data structure we use to calculate these distances, and Sections 6.2 and 6.3 give the two value functions tested in this paper.

6.1 Computing Distance Maps

For route planning, it is necessary to estimate the travel time between all pairs of waypoints, taking obstacles into consideration. For the evaluation function used by the driving algorithms, it is necessary to find the distance between the ship and the next waypoint, again accounting for obstacles.

The distances between every waypoint and every other non-obstacle point on the map are computed up-front. This can be done quickly and efficiently using a scanline floodfill algorithm: maps for the PTSP are represented as 2-dimensional arrays, where

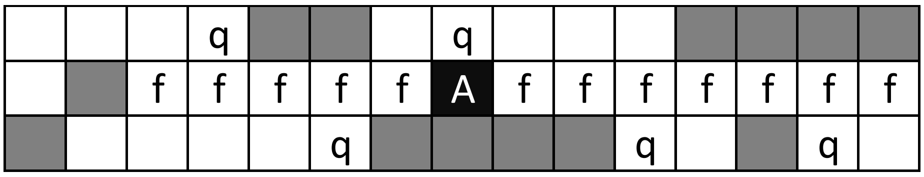


Figure 6: An example of the scanline flood fill algorithm. Cells coloured grey are unfillable. The cell marked A is the current cell. The cells marked f, on the same row as A, are filled. The cells marked q, on the rows directly above and below A, are enqueued.

each cell is either an obstacle or open space. This bitmap-like representation is particularly amenable to algorithms from computer graphics. Distance maps are computed using a modified scanline flood fill algorithm [21]; see Algorithm 1.

The resulting 2-dimensional array is called a *distance map*. Once these distance maps are computed, finding the distance between a waypoint and any other point (waypoint or ship position) can be looked up quickly in $O(1)$ time.

The algorithm initialises every entry in the array to $+\infty$, apart from the entry corresponding to waypoint i's position which is initialised to 0. The algorithm maintains a queue of cells from which to scan, beginning with waypoint i's position. From each cell, the algorithm scans to the left and to the right. For each scanned cell (x, y), a tentative distance $d_t(x, y)$ is computed as

$$d_t(x, y) = \min_{(x',y')} D[x', y'] + \sqrt{(x' - x)^2 + (y' - y)^2} \tag{5}$$

where (x', y') ranges over the orthogonal and diagonal neighbour cells of (x, y). The cell is considered *fillable* if $d_t(x, y) < D_i[x, y]$, i.e. if the tentative distance is less than the distance currently stored in D_i. During the scan, fillable cells are updated by setting $D_i[x, y] = d_t(x, y)$; the scan terminates upon encountering an unfillable cell. While scanning, the algorithm checks the cells immediately above and below the current row: upon transitioning from a region of unfillable cells to a region of fillable cells, the first fillable cell is enqueued. An example of this is shown in Figure 6.

Obstacle cells are always considered unfillable. Every obstacle is also surrounded with an unfillable $+$-shaped region whose radius is equal to the ship's radius. This effectively means that the distance map, and the algorithms that use it, ignore any spaces or corridors that are too narrow to accommodate the ship. A $+$-shaped rather than circular region is used purely for computational speed: for the purposes of removing narrow corridors, both are equally effective.

Whether a cell is fillable depends on the tentative distance, which in turn depends on the contents of the distance map surrounding the cell, a cell that has previously been filled or considered unfillable can become fillable again later on. This corresponds to a new path being found to a cell, shorter than the shortest previously known path. This is in contrast to the classical flood fill algorithm [21], where a previously filled pixel is never filled again.

Figure 7 shows an example of a distance map. Note that since distances are computed based on orthogonally and diagonally adjacent cells, the contours are octagonal rather than circular. Circular contours could be more closely approximated by enlarging the adjacency neighbourhood, but this implementation is a good tradeoff between

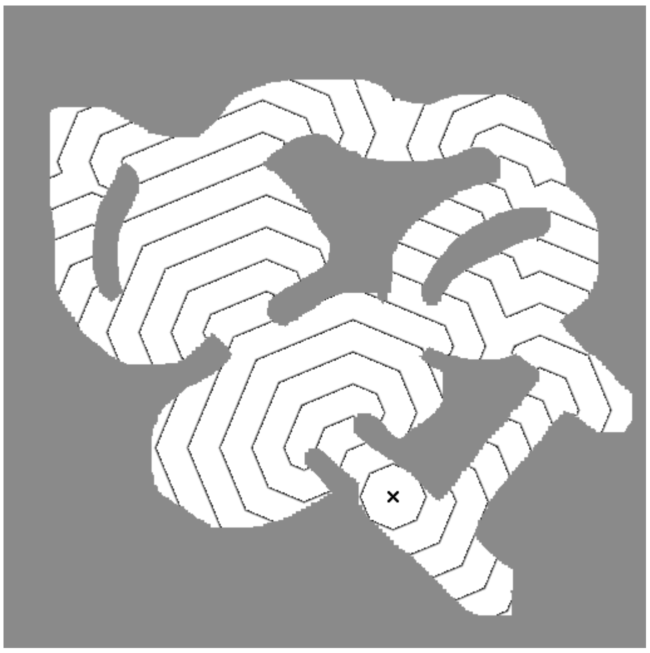


Figure 7: An example of a distance map for the point marked $\times$. This is a contour map: lines show regions where the distance map value is a multiple of 25.

speed and accuracy. The distance maps do not need to be completely accurate, merely accurate enough to serve as a guide for route planning and obstacle avoidance.

6.2 Myopic Evaluator

At any given point during the game, the route planner (Section 7) identifies the current target waypoint. Let s_r denote the fraction of the distance travelled towards the target waypoint, scaled so that $s_r = 0$ when the ship is at the previously collected waypoint (or at the start position if no waypoints have yet been collected) and $s_r = 1$ when the ship has reached the target waypoint. Here "distance" is the shortest path distance, looked up in the distance map for the target waypoint (Section 6.1).

The myopic evaluation for state s is calculated as follows:

$$V(s) = \begin{cases} s_r & \text{if the target waypoint is uncollected,} \\ \alpha_w > 1 & \text{if the target waypoint is collected.} \end{cases} \tag{6}$$

In other words, the state value is proportional to distance until the target waypoint is collected, with all post-collection states having the same value α_w. Optimising the distance in this way, instead of rigidly following the gradient descent paths suggested by the route planning phase, encourages the controller to take shortcuts to the next waypoint taking advantage of the physics of the game.

It is important that $\alpha_w > 1$, i.e. that post-collection states have a higher value than all pre-collection states, as this incentivises the search towards waypoint collection. Without this, the evaluation is very close at positions in the neighbourhood of a way-

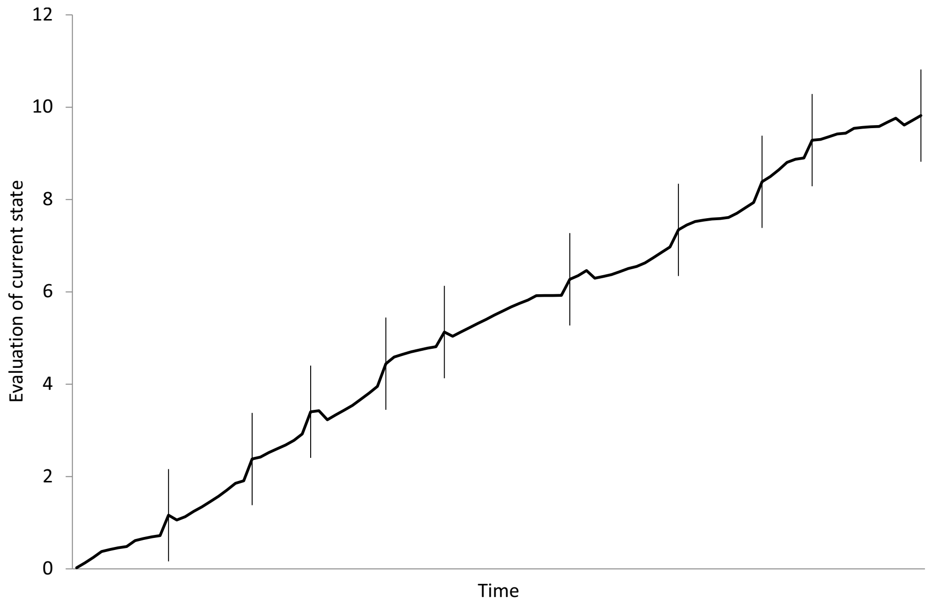


Figure 8: A plot of the evaluation of the current state by Equation 7 against time, for an instance of PTSP. Vertical lines denote states where a waypoint was collected whilst executing the previous macro-action. Note the jump in evaluation score at these states.

point, which often results in the ship waiting near the waypoint, or spiralling around it (usually getting closer with each rotation) but not actually passing through it.

6.3 Stepping Evaluator

The myopic evaluator is short-sighted in that it does not distinguish states after collection of the target waypoint. Another value function is defined to remedy this, which causes the ship to collect the current target waypoint and end up in a favourable position for collecting the waypoint after it.

The evaluation of the ship being in a particular state s is calculated as follows:

$$V(s) = \alpha_w s_w + \alpha_r s'_r + \alpha_s s_s \tag{7}$$

where s_w is the number of waypoints that have been collected so far along the route recommended by the route-planning phase (Section 7), s'_r is the proportion of the distance travelled from the last waypoint to the next target waypoint in state s according to the distance map (Section 6.1) and s_s is the speed of the ship, to encourage the steering algorithm to maintain momentum. The α values are weights that need to be tuned.

Note that s'_r is defined similarly to s_r for the myopic evaluator, with one crucial difference: s_r considers the target waypoint in the root state, whereas s'_r considers the target waypoint in state s. This is the distinction between the myopic and stepping evaluators: once the target waypoint is collected, the myopic evaluator considers all states to be equally good whereas the stepping evaluator begins optimising the distance to the next waypoint. Choosing $\alpha_r < \alpha_w$ means that there is a step (a discontinuous jump) in reward associated with collecting the next waypoint (see Figure 8).

The evaluation explicitly does not reward the agent for collecting waypoints early (out of route order). Otherwise the MCTS driver has a tendency to make detours to greedily collect waypoints, which generally turns out to be detrimental in the long term.

Since the amount of time taken is not directly represented in the evaluation, a different evaluation is used at terminal states when all waypoints are collected:

$$V(s) = \alpha_w W + \alpha_t (T_{\text{out}} W - t) \tag{8}$$

where W is the number of waypoints, T_{out} is the timeout between collecting waypoints, $t \leq T_{\text{out}} W$ is the total number of time steps taken to collect all the waypoints, and α_t is a parameter to tune. This ensures that terminal states have an evaluation of at least $\alpha_w W$, which is higher than that of any non-terminal states, and that terminal states which are reached in less time have a higher evaluation than terminal states which took more time.

The stepping evaluator worked well in the PTSP competition, but has a few flaws. Most notably, if the ship must turn in the opposite direction after collecting a waypoint, initially after collecting it the value function will be decreasing as the ship travels further away from the new target waypoint, and we will have $s_r < 0$. This occasionally results in the driving algorithm not collecting the waypoint, as states in which it is collected appear worse than states in which it is not. This situation can be seen in Figure 8 as a dip in reward immediately following the spike for collecting the waypoint; if the depth of the dip exceeded the height of the spike, the likelihood is that the controller would fail to collect the waypoint. Solving this problem requires careful balancing of the parameters α_w and α_r, to ensure collecting a waypoint always results in a net increase in value.

7 TSP Solvers

This section describes the methods tested for planning the order in which waypoints are visited. The ordering is crucial to performance, since a good ordering will result in fast and short routes and a bad ordering may be impossible to traverse (the controller will run out of time between waypoints). Further, the search techniques used for steering the ship do not plan far enough ahead to find good waypoint orderings. In all cases, the distance between waypoints refers to the floodfill distance taken from precomputed distance maps (Section 6.1) and not Euclidean distance.

7.1 Nearest Waypoint Now

The simplest "planner" tested does not plan ahead at all. It always instructs the steering algorithm to collect the nearest waypoint. The target waypoint may change on the path between two waypoints, when one becomes closer than another. This can lead to indecisive behaviour in the controller, the target waypoint may frequently change and no waypoints are collected. All the other planning methods avoid this issue by deciding a fixed ordering upfront.

7.2 Nearest Waypoint First Planner

This planner selects a waypoint ordering greedily, by choosing the closest waypoint at each step. In other words, the ith waypoint in the route is the closest waypoint (that does not occur earlier in the route) to the $(i-1)$th waypoint. These orderings may

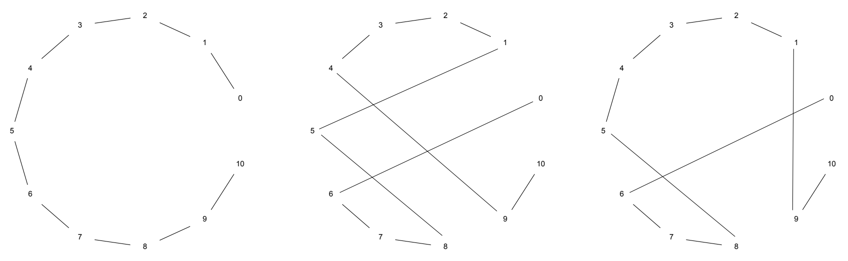


Figure 9: A Hamiltonian path and its two 3-opt moves.

not always be possible to traverse, since the greedy ordering may result in adjacent waypoints that are far from each other.

7.3 TSP Planner

All of the maps used in the experiments have 30, 40 or 50 waypoints. Solutions to 51-node min-weight Hamilton path problems can be found quickly using the greedy multiple fragment heuristic [22] and 3-opt local improvement [23, 24] and these are close enough to optimal for our purposes.

Multiple fragment [22] is a greedy heuristic for constructing a Hamiltonian path. The algorithm finds the set of edges in the path, iteratively adding the edge with minimal weight that does not invalidate the path so far (i.e. does not create a cycle or result in a node having a degree greater than 2).

3-opt [23] is a local search technique for refining a path such as that constructed by multiple fragment: a *3-opt move* consists of removing three edges from the path and adding three edges to obtain a new path; The 3-opt operator repeatedly applies 3-opt moves that reduce the cost of the path, until no such moves are possible. For each triple of edges in the path, two 3-opt moves are possible, as illustrated in Figure 9.

This planner uses multiple fragment and 3-opt, with edge weights computed by distance map, to compute the route.

7.4 Physics Enabled TSP Planner

The TSP planner assumes that the time taken to traverse a route is proportional to its length. As observed in Section 2.2, this is not true in PTSP: a long straight route with no sharp turns can often be traversed more quickly than a shorter but more winding route.

To identify sharp turns, it is useful to estimate the angles at which the ship will enter and leave each waypoint on a particular route. This is not simply the angle of a straight line drawn between one waypoint and the next, as this line may be obstructed.

Distance maps can be traversed to find a path from u to v. Let D_v be the distance map for waypoint v. Then a path is $p_0, p_1, \ldots, p_k$, where $p_0 = u$, $p_k = v$, and p_{i+1} is the neighbour of p_i for which $D_v[p_{i+1}]$ is minimal. These *distance map traversal paths*, although “shortest” with respect to a particular metric, are a poor approximation of the paths taken by the MCTS steering controller.

The *path direction* at u towards v, denoted $\overrightarrow{uv}$, is an approximation of the direction in which the ship leaves u when travelling towards v, or enters u when travelling from v. It is obtained by following the distance map traversal path from u to v until the

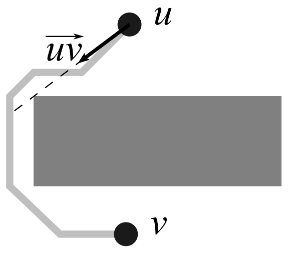


Figure 10: Computing the path direction $\overrightarrow{uv}$. The thick grey line is the distance map traversal path, according to v's distance map. The dotted line links u with the first point on the path such that this line is obstructed. The vector $\overrightarrow{uv}$ is the unit vector in the direction of this line.

first instance where the line between u and the current point p_i is obstructed. $\overrightarrow{uv}$ is taken to be the unit vector in the direction of $p_i - u$. This is illustrated in Figure 10. This process of stepping along the path until line-of-sight with the starting point is lost, rather than e.g. stepping along a fixed distance, ensures that the path directions do not suffer the same bias towards diagonal and orthogonal movement as the paths themselves and thus more closely approximate the direction of the ship (assuming the steering algorithm does not choose a path completely different to the distance map traversal path, which is not guaranteed). This process is similar to the *string-pulling* technique [25] often used in pathfinding for video games.

A "directness" heuristic is also introduced, which is based upon the principle that paths between two waypoints that have fewer obstacles should be preferred over paths which have many obstacles and tight turns. *Directness* is measured by calculating the ratio between the Euclidean distance and the path distance between two waypoints. If there are few obstacles this ratio will be small (tending towards 1), however if the path is long and complex, this ratio will be large. Hence a path is considered more indirect the more it deviates from a straight line in open space

3-opt normally seeks to minimise the sum of edge weights on the path. To account for the physical considerations described above, the aim is to minimise a cost function, incorporating terms that penalise sharp turns at waypoints and indirect paths between waypoints in addition to the sum of edge weights. *Multiple fragment* merely provides an initial guess to be refined by 3-opt, so little would be gained by modifying multiple fragment in a similar way.

For vertices u and v:

1. let $d(u, v)$ be the shortest path distance between u and v, computed using the distance map;
2. let $e(u, v)$ be the Euclidean distance between u and v;
3. let $\overrightarrow{uv}$ be path direction at u towards v.

Then the cost of a path $v_0, v_1, \ldots, v_n$, assuming that the ship is initially facing in direction $\mathbf{u}_0$, is

$$c(v_0,\ldots,v_n) = \sum_{i=1}^{n} d(v_{i-1},v_i) + \beta_p \sum_{i=1}^{n} \frac{d(v_{i-1},v_i)}{e(v_{i-1},v_i)} + \beta_w \left(-\overrightarrow{u_0} \cdot \overrightarrow{v_0 v_1} + \sum_{i=1}^{n-1} \overrightarrow{v_i v_{i-1}} \cdot \overrightarrow{v_i v_{i+1}} \right) \quad (9)$$

for constants β_w and β_p. The first term is the sum of edge weights. The term modified by β_p measures the directness of the path from one waypoint to the next as the ratio of the path distance and the Euclidean distance. The term modified by β_w measures the sharpness of the turns the ship needs to make when leaving the starting point and when travelling through each waypoint. If passing through a given waypoint does not require a change of direction, the incoming and outgoing vectors point in opposite directions and so their dot product is -1 (i.e. the cost is negative). If passing through the waypoint requires a $180°$ turn, the dot product is $+1$ (i.e. the cost is positive). β_p and β_w are parameters to be tuned.

3-opt uses this heuristic in the natural way: when considering whether to make a 3-opt move, the value of $c(v_0,\ldots,v_n)$ is considered in place of the total path length. Note that if $\beta_p = \beta_w = 0$, this planner is identical to the TSP planner without physics.

8 Experimental Setup

All the experiments described in this paper have been executed in the set 20 maps used to run the final evaluations of the IEEE CIG 2012 PTSP Competition, each map being played 10 times. These maps contain 30, 40 and 50 waypoints, which makes the problem much more challenging than the maps from the WCCI Competition (only 10 waypoints). Also, this set of maps permits a straightforward comparison with the other entries to the competition. Additionally, the machine used to run the experiments is the same server from which the competition was run (a dedicated Intel Core i5 server, 2.90GHz 6MB, and 4GB of memory), which provides reliable comparisons.

The experiments presented in this paper analyse the impact of each one of the following parameters:

- Search method: DFS, MC or MCTS. The parameter C for Equation 4 has been set to 1, a value determined empirically.
- Depth of simulations (D): Number of macro-actions used, set to 120, 24, 12, 8, 6 and 4.
- Macro-action length (T): each simulation depth is assigned to one of the following values for T: 1, 5, 10, 15, 20 and 30, respectively. Note that each pair tested (D_i,T_i) produces a look ahead in the future of $D_i \times T_i = 120$ steps (or single actions). This produces a fair comparison between algorithms that use different values for these two parameters.
- Function evaluator: Myopic Evaluator versus Stepping Evaluator.
- TSP Solver: Nearest Waypoint Now, Nearest Waypoint First Planner, TSP Planner and Physics-Enabled TSP Planner.

The experiments have been divided into two groups of tests. A preliminary batch of experiments was run in order to find a good driver for the PTSP, combining only the first four parameters described above. These tests aim to identify those drivers that are able to reach most of the waypoints of each map, using the simplest TSP solver prepared for this study (Nearest Waypoint First, see Section 7.1). This also allows us to discard some of the parameter values that perform worse, in order to focus the second batch of tests on the most promising parameter values, where the other three TSP solvers are tried.

Comparing two executions with several runs is not as trivial as it might seem. The simplest option would be to calculate the average of waypoints visited and the time taken to do so. However, this might lead to some unfair situations. An example would be to have two drivers, A and B, that obtain different amounts of waypoints on average ($w_a = 30$, $w_b = 29$) with their respective times ($t_a = 2500$, $t_b = 1800$). Following the game definition of which solver is better (as described in Section 2), A should be considered to be better than B because $w_a > w_b$. However, one could argue that B is better as it is much faster and the difference between waypoints is not too big. Therefore, two different measures are taken to compare results:

- *Efficacy*: number of waypoints visited, on average. This value is to be maximized.
- *Efficiency*: the ratio t/w, that represents an average of the time needed to visit each waypoint. The problem with this ratio is that it does not scale well when the amount of waypoints is very small, or even 0. Therefore, only matches where all waypoints have been visited are considered for this measure. The smaller this value, the better the solution provided.

9 Results and Analysis

This section presents the results of all the experiments performed in this research. These results are also available in a document[1] and in the PTSP Competition website[2]. The document contains links to each one of the runs, which detail statistical information and the result of each single match. Additionally, the page with the run details contains links that allows all matches played to be watched in a Java applet.

9.1 Macro-action length and function evaluator

The first batch of experiments analyses the effects of using different macro-action lengths and value functions. In this case, all controllers employ the Nearest Waypoint Now TSP solver (as described in Section 7.1). The goal is to be able to identify the best attributes for the driver.

Figures 11 and 12 show the number of waypoints visited using the Myopic and Stepping evaluation functions respectively, measuring the *efficacy* of the solvers. Both pictures include a horizontal line showing the maximum average of waypoints achievable (39), that comes from the distribution of waypoints in the maps tested (7 maps with 30 waypoints, 8 maps containing 40 and 5 maps with 50).

These two figures show that the Myopic approach performs slightly worse than the Stepping evaluator function. For both functions, the number of waypoints visited

[1]`https://www.dropbox.com/sh/ksxa53qtm5cjsmj/A_wxxMrLGE`
[2]`http://www.ptsp-game.net/bot_rankings_advanced_f1.php`

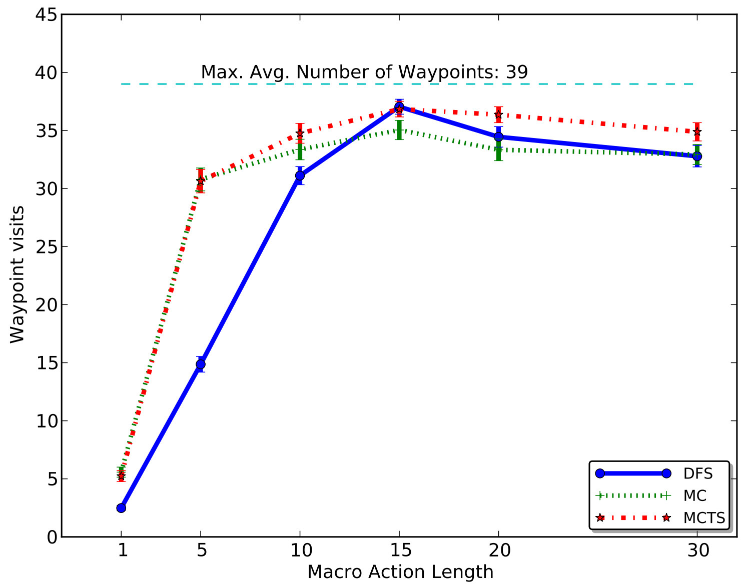


Figure 11: Number of waypoint visits with Myopic function evaluation and Nearest Waypoint Now TSP solver.

increases when the length of the macro-actions reaches a high value (10 to 20). Table 1 shows that the number of waypoints visited decreases slightly when the macro-action length is increased to 30. This phenomenon happens for the three search methods presented in this paper, suggesting that the reduction of the problem achieved by the inclusion of macro-actions leads to better performance.

In general, DFS performs worse than the other algorithms, and even in those cases where the length of the macro-actions is small (1 and 5), better results are obtained with MC and MCTS. It is also worthwhile to mention that none of the algorithms is able to reach an optimal efficacy (39 waypoints). In other words, all methods fail to catch all waypoints in at least one of the 200 matches played, which shows the complexity of the game.

The *efficiency* of the controllers is calculated as the average of the efficency ratio (time/waypoints) in those matches where all waypoints are visited. Figures 13 and 14 show these measurements for the Myopic and Stepping function evaluators respectively. As explained in Section 8, this ratio can be misleading if the number of waypoints visited is far from the amount of them in the maps. For this reason, the macro-action length of 1 is not included in these pictures.

The ratios obtained (the smaller, the better) show again that MC and MCTS methods achieve better results than DFS, and these results are better when the length of the macro-actions is high (10 to 20). However, as shown in Table 1, raising the value of T to 30 provides significantly worse results. In the comparison between the Myopic and Stepping function evaluators, better efficiency is obtained also with the Stepping evaluator, and in this case the difference is more significant (about 30 points).

Table 1 shows the numeric results of the tests, for all configurations tested, includ-

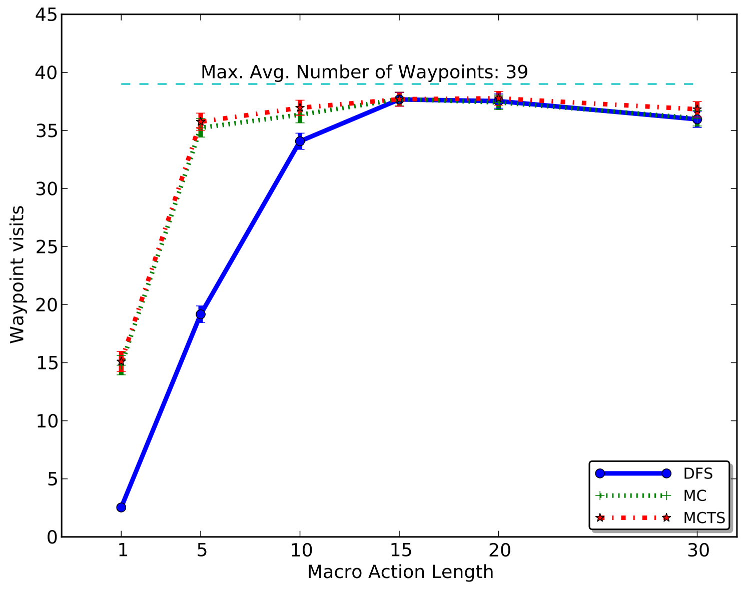


Figure 12: Number of waypoint visits with Stepping function evaluation and Nearest Waypoint Now TSP solver.

ing the standard error of the averages obtained.

One point of interest is the ratio obtained when the length of the macro-action is 1: four out of the six configurations are unable to provide reasonable results. Only MC and MCTS, using the Stepping evaluator function, are able to visit at least once all the waypoints scattered around the maps tested. Another interesting result is that MCTS achieves the best performance for almost all combinations of macro-action length and value function. MC only gets slightly better results in two cases: number of waypoints visited, using Stepping function evaluator, for macro-action lengths of 15 and 20.

In many cases, the results obtained by each algorithm are very similar to each other. For this reason, the Mann-Whitney-Wilcoxon test (MW-test) has been performed in order to confirm if there is a statistical difference in those cases where the results are alike. For the sake of space, not all MW-test results are included in this paper, but the most interesting ones are explained here: the tests show that there is statistical difference between MCTS and the other two methods for 15 and 20 macro-action lengths using the Stepping evaluator (for $T = 15$, p-values of 0.109 and 0.001 against DFS and MC respectively. 5×10^{-6} and 1.5×10^{-6} for $T = 20$). In other words, this suggests that the lower ratio shown in Figure 14 responds to a truly better performance of MCTS over MC and DFS in these cases. Regarding efficacy, these methods provide the same results in the scenarios mentioned.

9.2 TSP Solvers

After the analysis performed in the previous section, some values for the parameters tested are discarded in order to focus on those that provide better results. The goal at

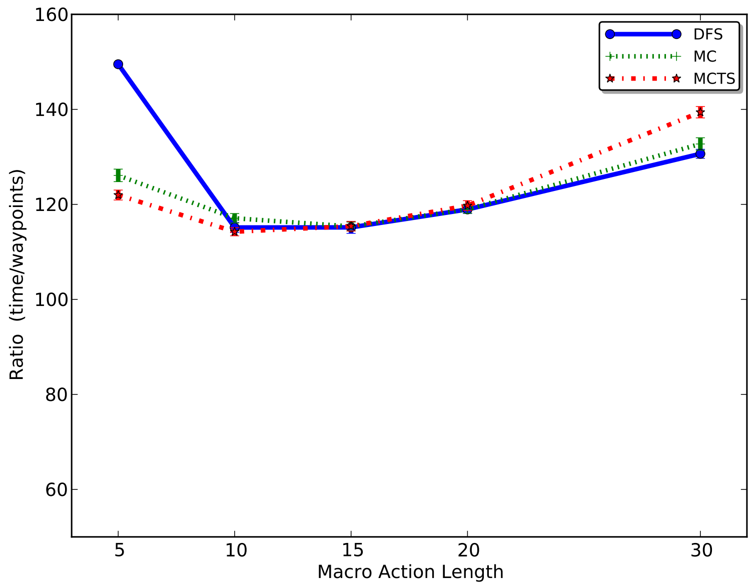


Figure 13: Ratio time/waypoints with Myopic function evaluation and Nearest Waypoint Now TSP solver.

this stage is to evaluate how the different TSP solvers affect the controllers tested. In this case, the parameters are configured as follows:

- Search method: DFS, MC or MCTS.
- Depth of simulations (D): 12, 8 and 6.
- Macro-action length (T): 10, 15 and 20.
- Function evaluator: only Stepping Evaluator.

Table 2 shows the results obtained with all TSP solvers and these parameters.

Several observations can be made about the performances shown in these tests. First, the overall best TSP solver is the Physics Enabled Flood Fill Planner, obtaining better performance in terms of waypoints visited and ratio of visits. This result confirms the hypothesis drawn in Section 2.2: better performance is obtained when the physics of the game are taken into consideration when planning the order of waypoints to visit.

For the best TSP planner, MCTS behaves better than both MC and DFS in the two measures taken. Regarding the macro-action length, a value of 15 provides the best result in ratio of waypoints visited, while 20 seems to be the best choice for the efficacy of the controller. Actually, the controller that obtained first place in the PTSP competition was the configuration using MCTS, Physics Enabled Flood Fill Planner and $T = 15$.

Again, statistical tests have been calculated between the most interesting values. For instance, Flood Fill Planner and Physics Enabled Flood Fill Planner have been

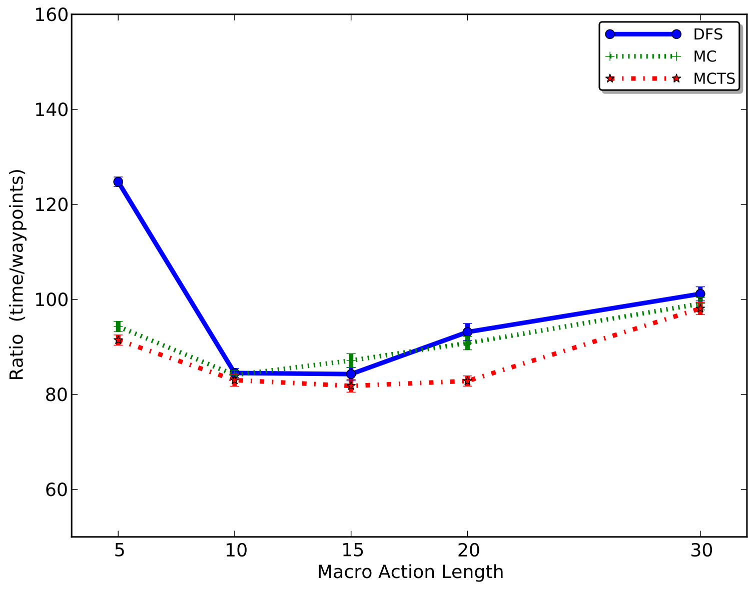


Figure 14: Ratio time/waypoints with Stepping function evaluation and Nearest Waypoint Now TSP solver.

compared for MCTS with T values of 15 and 20. The MW-Test provides values of 0.015 and 0.052, respectively, showing a high confidence in the case of $T = 15$ and only a fair indication that the distributions are different for $T = 20$.

Another interesting result is that Nearest Waypoint Now solver seems to produce better results than the planner version of the same algorithm. It can be surprising that a planned route of waypoints behaves worse than a version that just pursues the closest waypoint at any time, but the explanation is again in the physics of the game. The Nearest Waypoint Now TSP solver only takes into account the distances between waypoints, but ignores the inertia and speed of the ship while travelling around the maze. This inertia could take the ship closer to another waypoint (which may not be the next one to visit in the planned version) but obviously, it could be faster to visit this one instead of changing the course to follow the plan. A better solution than the one provided by the Nearest Waypoint Now solver is obtained with the Physics Enabled Flood Fill Planner (80.10 versus 81.78, MW-test p-value of 3.07×10^{-5}), using $T = 15$ which seems to be the best overall value for the macro-action length.

9.3 Analysis by map

An interesting comparison between the techniques presented in this paper is the map by map comparative. It is possible that some methods behave better in distinct maps, depending on how the obstacles are distributed around the maze.

A score system that stresses the differences between maps is the one used in the PTSP Competition (see Section 2.3). This ranking scheme awards points for the rankings on each of the maps where the controllers are evaluated. The final winner is the

		DFS		MC		MCTS	
T	Value Function	Myopic	Stepping	Myopic	Stepping	Myopic	Stepping
1	Visits	2.48 ± 0.21	2.54 ± 0.19	5.58 ± 0.43	14.79 ± 0.87	5.24 ± 0.47	**15.10 ± 0.87**
	Ratio	–	–	–	99.88 ± 1.11	–	**93.14 ± 0.29**
5	Visits	14.87 ± 0.67	19.17 ± 0.72	30.79 ± 0.98	35.23 ± 0.79	30.64 ± 1.01	**35.76 ± 0.74**
	Ratio	149.50 ± 0.39	124.75 ± 1.02	126.10 ± 1.32	94.28 ± 1.09	121.96 ± 1.05	**91.43 ± 1.08**
10	Visits	31.11 ± 0.78	34.07 ± 0.70	33.36 ± 0.88	36.35 ± 0.69	34.76 ± 0.85	**36.95 ± 0.65**
	Ratio	115.13 ± 1.03	84.50 ± 0.96	117.08 ± 0.99	84.08 ± 0.95	114.24 ± 0.86	**83.00 ± 1.29**
15	Visits	37.05 ± 0.64	37.67 ± 0.59	35.04 ± 0.83	**37.70 ± 0.59**	36.84 ± 0.67	37.69 ± 0.60
	Ratio	115.15 ± 1.26	84.27 ± 1.41	115.36 ± 0.92	87.12 ± 1.45	115.41 ± 1.02	**81.78 ± 1.32**
20	Visits	34.45 ± 0.88	37.53 ± 0.61	33.34 ± 0.94	**37.41 ± 0.62**	36.36 ± 0.69	37.77 ± 0.60
	Ratio	118.96 ± 0.85	93.12 ± 1.79	118.96 ± 0.82	90.80 ± 1.41	119.74 ± 1.03	**82.82 ± 1.07**
30	Visits	32.78 ± 0.91	35.96 ± 0.69	32.93 ± 0.85	36.05 ± 0.67	34.88 ± 0.79	**36.82 ± 0.66**
	Ratio	130.64 ± 0.95	101.17 ± 1.48	132.71 ± 1.31	99.09 ± 1.31	139.4 ± 1.20	**98.08 ± 1.29**

Table 1: Waypoint visits and ratio for different algorithms and macro-action lengths, using Nearest Waypoint Now TSP solver. Showing average with standard error, bold for best result in each row.

		Nearest Waypoint Now		Nearest Waypoint First Planner		Flood Fill Planner		Physics En. Flood Fill Planner	
Method	T	Visits	Ratio	Visits	Ratio	Visits	Ratio	Visits	Ratio
DFS	10	34.07 ± 0.70	84.50 ± 0.96	32.83 ± 0.84	88.89 ± 0.92	30.49 ± 0.99	90.90 ± 1.77	31.13 ± 1.10	84.72 ± 1.14
	15	37.67 ± 0.59	84.27 ± 1.41	34.85 ± 0.87	97.61 ± 0.95	34.59 ± 0.88	88.33 ± 1.54	34.52 ± 0.85	85.54 ± 1.37
	20	37.53 ± 0.61	93.12 ± 1.79	29.50 ± 1.10	106.55 ± 0.82	28.34 ± 1.06	89.55 ± 0.85	30.36 ± 1.08	89.61 ± 1.05
MC	10	36.35 ± 0.69	84.08 ± 0.95	33.65 ± 0.94	101.71 ± 0.93	27.91 ± 1.10	96.89 ± 1.05	30.80 ± 1.11	84.82 ± 0.96
	15	37.70 ± 0.59	87.12 ± 1.45	34.54 ± 0.92	97.29 ± 0.86	29.89 ± 1.04	87.85 ± 0.79	32.02 ± 1.03	87.54 ± 1.00
	20	37.41 ± 0.62	90.80 ± 1.41	33.09 ± 1.03	99.72 ± 0.91	28.87 ± 1.06	92.45 ± 1.02	30.32 ± 1.12	89.84 ± 1.07
MCTS	10	36.95 ± 0.65	83.00 ± 1.29	35.24 ± 0.82	92.52 ± 0.88	30.50 ± 1.06	86.86 ± 1.14	34.98 ± 0.91	80.74 ± 1.01
	15	37.69 ± 0.60	81.78 ± 1.32	37.48 ± 0.64	87.76 ± 0.96	34.57 ± 0.93	82.84 ± 1.12	37.73 ± 0.65	**80.10 ± 1.29**
	20	37.77 ± 0.60	82.82 ± 1.07	37.62 ± 0.63	92.36 ± 0.98	37.44 ± 0.69	92.02 ± 1.74	**38.34 ± 0.62**	89.11 ± 1.76
Average		37.01 ± 0.21	85.82 ± 0.53	34.31 ± 0.29	95.68 ± 0.4	31.4 ± 0.33	89.46 ± 0.53	33.35 ± 0.32	85.56 ± 0.47

Table 2: Waypoint visits and ratio for the best macro-action lengths and Stepping evaluator, comparing all TSP solvers. Showing average with standard error, bold for overall best result in visits and ratio.

controller that achieves more points in total. For every map, the points are awarded as in the Formula One Championship: 25 points for the best controller, 18 points for the second best, 15 for the third and $12, 10, 8, 6, 4, 2$ and 1 for the bots ranking from fourth to tenth, respectively. The distribution of points in this scheme highly awards the best controller on each map.

Table 3 shows the points achieved in each map, following the PTSP competition ranking scheme. The controllers shown are the ten best bots ranked this way. It is interesting to see that the best controller is still the one submitted to the PTSP Competition that ranked first.

Controller vs. Map	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**Total**
MCTS-8-15-Stepping-PEFF	**25**	**25**	**25**	6	**25**	4	**25**	18	**25**	18	18	10	**25**	**25**	18	**25**	**25**	0	**25**	2	**369**
MCTS-8-15-Stepping-Near	15	15	4	**25**	15	18	0	12	10	4	6	**25**	12	8	6	15	8	8	2	**25**	233
MCTS-12-10-Stepping-PEFF	18	18	18	0	18	0	18	8	18	**25**	**25**	2	0	1	0	18	18	0	0	6	211
MCTS-6-20-Stepping-PEFF	0	8	15	0	6	0	15	4	12	12	12	1	18	6	0	12	15	**25**	18	0	179
MCTS-8-15-Stepping-FFNP	10	0	8	10	2	6	12	15	4	6	8	0	10	15	**25**	8	4	0	0	0	143
MCTS-12-10-Stepping-Near	2	6	6	18	8	**25**	0	0	0	0	0	15	15	0	2	2	6	0	6	18	129
DFS-4-15-Stepping-PEFF	8	12	10	0	10	0	10	10	15	15	15	0	0	0	0	10	0	0	8	0	123
MCTS-6-20-Stepping-Near	4	10	12	8	0	15	0	0	0	0	0	18	0	4	0	0	1	0	0	12	84
MC-12-10-Stepping-PEFF	0	0	0	0	12	0	6	**25**	0	0	0	0	4	10	15	0	10	0	0	0	82
DFS-4-15-Stepping-Near	1	2	0	12	1	8	0	0	0	8	0	6	0	0	10	4	0	0	10	15	77

Table 3: This table presents the rankings of the best controllers following the point award scheme of the PTSP competition. It shows ten controllers (rows of the table) evaluated on 20 maps (columns). The names of the controllers indicate (in this order): method, number macro-actions, macro-action length, evaluation function and TSP solver (FFNP: Flood Fill Planner; PEFF: Physics-Enabled Flood Fill Planner; Near: Nearest Waypoint Now).

However, it is not able to achieve the best results in all maps: in map 18, it gets 0 points (note that this does not mean that it does not visit any waypoint. It is just worse than the tenth controller in this particular map). The best result in this map is obtained by the same algorithm but using $T = 20$ as macro-action length, which suggests that for certain maps this parameter is crucial to obtain a good performance.

The PTSP Competition scheme favours those controllers that visit more waypoints, with the time spent relegated as a tie breaker when the visits are the same. This is the reason why in these rankings an algorithm that uses the Nearest Waypoint Now solver qualifies second (as seen in Table 2, the efficacy of this TSP solver is high).

Figure 15 shows the waypoint ratios per map, which allows a comparison of the efficiency of the different controllers. For the sake of clarity, this picture shows only the best four controllers according to the rankings of this section. As can be seen, the best approach generally needs less time steps to visit waypoints, while the second controller often uses more time for this. Some exceptions are maps 4 and 6, where the controller using the Nearest Waypoint Now solver provides better solutions. This is strongly related to the results shown for these maps in Table 3, where the best controller does not achieve the best performance.

Finally, maps 18 and 19 are missing some of the bars for some controllers. These are the cases where the bots were not able to visit all waypoints in any of the matches played in these maps, and therefore the ratio calculation is not very descriptive. In these cases, the controllers achieved 0 points in the maps, as shown in Table 3.

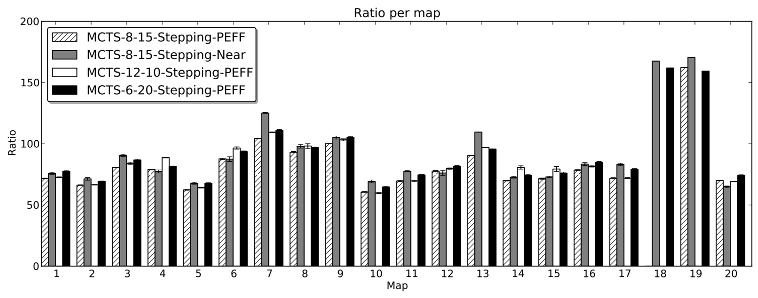


Figure 15: Ratio time/waypoints per map. The smaller the bar, the faster the controller visits waypoints. In maps 18 and 19, not all algorithms are able to reach all the waypoints in any of the games played.

9.4 Performance in the PTSP Competition

As was mentioned before, the configuration that provided the best results in this study (MCTS, $T = 15$, Stepping evaluator function and Physics Enabled Flood Fill Planner) was the winner of the CIG PTSP Competition. It would be interesting to see if any of the other configurations tested here would have won the competition anyway, using the results obtained in this research. The reasoning behind this test is as follows: in many competitions, the controllers and their heuristics are tweaked until they obtain the best possible performance. One might question, therefore, whether the winner of this (or any other) competition is providing a superior method, or only a very good heuristic that could work with different AI techniques.

The document referenced at the beginning of Section 9 contains links to the potential results that would have been obtained if each potential controller had been submitted *instead* of the one that was actually sent. The following lines summarize the statistics drawn from these potential submissions:

- None of the controllers tested would have won the competition using the Myopic function evaluator.
- Out of the configurations tested, all controllers which used the Physics Enabled Flood Fill Planner would still have won the competition.
- Out of those controllers that used Flood Fill Planner (no physics), all but one would have finished the competition in first place. Only MC with $T = 10$ would not win, ranking second instead.
- Only the MCTS controllers that use the Nearest Waypoint Planner TSP solver would have finished in first position in all cases tested.
- For the Nearest Waypoint Now TSP solver, only 14 out of 36 (38%) would still win.
- In general, 38 out of the 63 different tests performed (60%) would have won the competition.

Of course, these measures are illustrative, and they might well be different had the other competitors submitted different controllers, but the test still allows the drawing of some conclusions. For instance, these numbers suggest that MCTS, with the appropriate evaluation function, is the best algorithm submitted so far to the PTSP competition. The values of T to discard are 1, 5 and 30, but any of the other values (10, 15 and 20) produce results that are good enough to win the competition.

10 Conclusions and Future Work

This paper presents a study of the algorithm that won the PTSP IEEE CIG 2012 competition, some variations of it, and how the different values for the algorithm's constituents affect the overall performance of the controller. This performance is measured in terms of the efficacy (or how many waypoints are visited) and the efficiency (the average time steps needed to visit waypoints) of the controllers.

One of the first conclusions that this study offers is the fact that using macro-actions in the PTSP improves the quality of the solutions enormously. This is especially interesting considering the rather simplistic *repeat-T-times* nature of the macro-actions used. The macro-action lengths that provided good performance are values from 10 to 20, obtaining the best results with $T = 15$.

It is also worthwhile mentioning that the design of an appropriate value function, especially for games like the PTSP where the real-time constraints usually prevent the algorithm from reaching an end game state, is crucial in order to obtain good results. In this particular case, a simple Myopic state evaluator behaves poorly compared with a more sophisticated Stepping evaluator.

Another aspect highlighted in this paper is the dependency between the order of waypoints and the driving style of the bot. In other words, how the physics of the game affect the optimal sequence of waypoints to follow. It has been shown in the results obtained that the TSP solver that considers the physics of the game achieves significantly better results than the others.

Finally, MCTS has been shown to behave better than the other two algorithms compared in this paper, MC and DFS. Of particular significance is the fact that any of the MCTS variants tested in this research, using the Stepping evaluator function and the appropriate macro-action lengths, would have won the CIG PTSP Competition.

This research can be extended in several ways. For instance, it would be interesting to investigate how the shape of the maps affects the performance of the algorithms, and under what specific conditions some macro-action lengths seem to work better. Additionally, as exposed in this paper, the design of the macro-actions is just a simple repetition of actions. It would be interesting to develop more sophisticated macro-actions and compare their performance. Another possibility is to further improve the algorithms presented: by looking at the results and games played, it is clear that the performance is not optimal. Combining MCTS with other techniques, such as evolutionary algorithms or TD-learning, could improve the results shown in this study.

Finally, another interesting test would be to reduce the time budget further and test how this affects the quality of the solutions obtained. Initial experiments show that at least MCTS still produces high quality solutions when the budget is reduced to 10ms. Nevertheless, further testing is needed to understand how this limitation affects the search methods presented here, as well as finding the minimum budget time that allows these techniques to keep providing good results in this and similar domains, where decisions must be taken within a few milliseconds.

Acknowledgments

This work was supported by EPSRC grants EP/H048588/1 and EP/H049061/1, both under the project entitled “UCT for Games and Beyond”.

References

[1] D. Perez, P. Rohlfshagen, and S. Lucas, “Monte Carlo Tree Search: Long Term versus Short Term Planning,” in *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 2012, pp. 219 – 226.

[2] R. Coulom, “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search,” in *Proc. 5th Int. Conference Comput. and Games*, Turin, Italy, 2006, pp. 72–83.

[3] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo planning,” *Euro Conference in Machine Learning*, vol. 4212, pp. 282–293, 2006.

[4] C.-S. Lee, M.-H. Wang, G. M. J.-B. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong, “The Computational Intelligence of MoGo Revealed in Taiwan’s Computer Go Tournaments,” *IEEE Transacttions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 73–89, 2009.

[5] 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*, vol. 4:1, pp. 1–43, 2012.

[6] S. Samothrakis, D. Robles, and S. M. Lucas, “A UCT Agent for Tron: Initial Investigations,” in *Proceedings of IEEE Symposium on Computational Intelligence and Games*, Ireland, 2010, pp. 365–371.

[7] N. G. P. D. Teuling and M. H. M. Winands, “Monte-Carlo Tree Search for the Simultaneous Move Game Tron,” in *Computer Games Workshop at ECAI*, Montpellier, France, 2012, pp. 126–141.

[8] N. Ikehata and T. Ito, “Monte-Carlo Tree Search in Ms. Pac-Man,” in *Proceedings of IEEE Conference on Computational Intelligence and Games*, 2011, pp. 39–46.

[9] C. Zhongjie, D. Zhang, and B. Nebel, “Playing Tetris Using Bandit-Based Monte-Carlo Planning,” in *Proceedings of AISB 2011 Symposium: AI and Games*, 2011, pp. 18–23.

[10] D. Perez, P. Rohlfshagen, and S. Lucas, “Monte-Carlo Tree Search for the Physical Travelling Salesman Problem,” in *Proceedings of EvoApplications*, 2012, pp. 255–264.

[11] E. J. Powley, D. Whitehouse, and P. I. Cowling, “Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling Salesman Problem,” in *Proceedings of IEEE Conference on Computational Intelligenece and Games*, Spain, 2012, pp. 234–241.

[12] A. McGovern and R. S. Sutton, “Macro-actions in reinforcement learning: an empirical analysis,” The University of Massachusetts - Amherst, Tech. Rep., 1998.

[13] S. Ontañón, “Experiments with game tree search in real-time strategy games,” *CoRR*, vol. abs/1208.1940, 2012.

[14] R.-K. Balla and A. Fern, “UCT for Tactical Assault Planning in Real-Time Strategy Games,” in *Proceedings of the 21st International Joint Conference on Artificial Intelligence.*, 2009, pp. 40–45.

[15] B. E. Childs, J. H. Brodeur, and L. Kocsis, “Transpositions and Move Groups in Monte Carlo Tree Search,” in *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2008, pp. 389–395.

[16] G. Van Eyck and M. Müller, “Revisiting move groups in monte-carlo tree search,” *Advances in Computer Games*, pp. 13–23, 2012.

[17] E. J. Powley, D. Whitehouse, and P. I. Cowling, “Determinization in Monte-Carlo Tree Search for the card game Dou Di Zhu,” in *Proceedings Artificial Intelligence and the Simulation of Behaviour*, York, United Kingdom, 2011, pp. 17–24.

[18] 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.

[19] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time Analysis of the Multiarmed Bandit Problem,” *Mach. Learn.*, vol. 47, no. 2, pp. 235–256, 2002.

[20] L. Kocsis, C. Szepesvári, and J. Willemson, “Improved Monte-Carlo Search,” Univ. Tartu, Estonia, Tech. Rep. 1, 2006.

[21] H. Lieberman, “How to color in a coloring book,” *ACM SIGGRAPH Computer Graphics*, vol. 12, no. 3, pp. 111–116, 1978.

[22] J. L. Bentley, “Experiments on Traveling Salesman Heuristics,” in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990, pp. 91–99.

[23] S. Lin, “Computer solutions of the traveling salesman problem,” *Bell System Technical Journal*, vol. 44, pp. 2245–2269, 1965.

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

[25] G. Snook, “Simplified 3D Movement and Pathfinding Using Navigation Meshes,” in *Game Programming Gems*. Charles River Media, 2000, pp. 288–304.

Algorithm 1 Scanline flood fill algorithm for computing distance maps. For conciseness, array bound checks are omitted.

```
function ComputeDistanceMap($M, x_0, y_0$)
    create a 2-D array $D$ with the same dimensions as $M$
    for each coordinate pair $(x, y)$ in the map do
        if $M[x, y]$ is a wall, or $M[x \pm i, y]$ then
            $D[x, y] \leftarrow -\infty$
        else if $\exists i, 0 < i \leq r_{\text{ship}} : M[x, y \pm i]$ is a wall then
            $D[x, y] \leftarrow -\infty$
        else if $x = x_0$ and $y = y_0$ then
            $D[x, y] \leftarrow 0$
        else
            $D[x, y] \leftarrow +\infty$
    create a queue $q$, and push $(x_0, y_0)$ onto it
    while $q$ is not empty do
        pop $(x, y)$ from $q$
        if IsFillable$(x, y)$ then
            $D[x, y] \leftarrow$ MinDist$(x, y)$
            if IsFillable$(x, y - 1)$ then
                push $(x, y - 1)$ onto $q$
            if IsFillable$(x, y + 1)$ then
                push $(x, y + 1)$ onto $q$
            Scan$(x, y, -1)$
            Scan$(x, y, +1)$
    return $D$

function Scan($x, y, \delta$)
    for $x' = x + \delta, x + 2\delta, x + 3\delta, \dots$ do
        if IsFillable$(x', y)$ then
            $D[x', y] \leftarrow$ MinDist$(x', y)$
            if IsFillable$(x', y - 1) \wedge$
            $\neg$IsFillable$(x' - \delta, y - 1)$ then
                push $(x', y - 1)$ onto $q$
            if IsFillable$(x', y + 1) \wedge$
            $\neg$IsFillable$(x' - \delta, y + 1)$ then
                push $(x', y + 1)$ onto $q$

function MinDist$(x, y)$
    $\eta \leftarrow \{(x', y') : x' \in \{x - 1, x, x + 1\},$
        $y' \in \{y - 1, y, y + 1\}, (x', y') \neq (0, 0)\}$
    return $\min_{\substack{(x', y') \in \eta \\ D[x', y'] > -\infty}} D[x', y'] + \sqrt{(x' - x)^2 + (y' - y)^2}$
```