# Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling Salesman Problem

Edward J. Powley, *Member, IEEE,* Daniel Whitehouse, *Graduate Student Member, IEEE,*
and Peter I. Cowling, *Member, IEEE*

***Abstract*—We present a controller for the *Physical Travelling Salesman Problem (PTSP)*, a path planning and steering problem in a simulated continuous real-time domain. Our approach is hierarchical, using domain-specific algorithms and heuristics to plan a coarse-grained route and *Monte Carlo Tree Search (MCTS)* to plan and steer along fine-grained paths. The MCTS component uses *macro-actions* to decrease the number of decisions to be made per unit of time and thus drastically reduce the size of the decision tree. Results from the 2012 WCCI PTSP Competition show that this approach significantly and consistently outperforms all other submitted AI controllers, and is competitive with strong human players. Our approach has potential applications to many other problems in movement planning and control, including video games.**

## I. Introduction

The *physical travelling salesman problem (PTSP)* extends the travelling salesman problem (TSP) [1] into a continuous real-time domain. In TSP the agent must choose a subset of the edges of a weighted graph which forms a tour covering all vertices, aiming to minimise the total weight. In PTSP the agent moves on the 2-D plane under Newtonian physics, and must find a path passing through all waypoints, aiming to minimise the time taken to travel this path. In this paper we present the details of our submission to the 2012 PTSP Competition[1] [2], which uses a number of novel techniques that may be applicable to a variety of other domains. In the WCCI 2012 PTSP competition, our controller significantly outperformed all other submissions to the AI competition. Our controller also performed strongly in the human versus AI competition, achieving second place overall and either obtaining or being within a small margin of the best score on each map.

*Monte Carlo Tree Search (MCTS)* is a game tree search algorithm that has yielded recent successes in games where traditional heuristic-based search techniques (such as depth-limited minimax) perform poorly [3]. Go is a prime example of such a game [4], [5], although MCTS has been applied to many other domains [6]. In its most basic form MCTS does not require expert domain knowledge, although it can often be enhanced with such information if it is available. MCTS is *anytime*: rather than requiring a fixed amount of CPU time, the algorithm attempts to make the best possible use of whatever CPU time is available.

PTSP is real-time in the sense that the AI agent must make decisions within strict time limits. As the environment is simulated on a computer, it is not real-time in the sense of continuous time: the game proceeds in discrete time steps, one every 40 milliseconds. However simply searching the decision tree induced by making one decision per time step would be ineffective: the tree has a branching factor of 6 and a depth varying between 1000 and 10000, making $10^{1500}$ a conservative estimate for its size. This dwarfs the tree size of even the most complex turn-based game.

The majority of applications of MCTS are to turn-based games such as board games. However MCTS has been applied to some real-time games, such as Tron [7], [8], Pac-Man [9], [10], [11] and others. Two common themes are present in these applications: the anytime nature of MCTS is clearly beneficial when decision time is severely limited; however, pure MCTS with random simulations tends to be weaker than e.g. MCTS modified with early cutoff and heuristic evaluation. Perez et al [12] apply MCTS to a simplified version of PTSP, finding that embedding of basic heuristic knowledge significantly improves performance, but even so the agent lacks the ability to plan ahead and instead takes a greedy approach.

This paper presents a novel MCTS-based approach to the physical travelling salesman problem. Our approach has three key features. Firstly, the agent does not choose actions but *macro-actions*, which in this case specify a single action to be repeated over multiple time steps. The use of macro-actions yields a linear reduction in tree depth and thus an exponential reduction in tree size. Secondly, the MCTS search is depth-limited, with a heuristic evaluation function for nonterminal states. This allows MCTS to achieve short-term goals (i.e. visiting the next waypoint) when the long-term goal (i.e. having visited all waypoints) is several hundred macro-actions into the future. The idea of depth limited MCTS was also proposed by Samothrakis et al [9] for the game of Ms Pac-Man. Thirdly, the heuristic evaluation is not static but is informed by a higher-level planning phase. The evaluation function rewards waypoints being collected in a particular order, and uses map features to guide the agent towards the next waypoint. Both the ordering and the map features are computed by the high-level planner rather than being hard-

E.J. Powley, D. Whitehouse and P.I. Cowling are with the Artificial Intelligence Research Centre, School of Computing, Informatics and Media, University of Bradford, UK. From September 2012 they will all be at the University of York, UK. E-mail: `e.powley@bradford.ac.uk`, `d.whitehouse1@student.bradford.ac.uk`, `peter.cowling@york.ac.uk`.

This work is funded by grant EP/H049061/1 of the UK Engineering and Physical Sciences Research Council (EPSRC).

[1]The 2012 PTSP Competition (held at WCCI 2012 and CIG 2012) is organized by Diego Perez, Philipp Rohlfshagen, David Robles and Simon Lucas from The University of Essex.

coded at design time.

This hierarchical approach allows us to use different AI techniques for higher-level planning (finding the best order in which to visit the waypoints) and lower-level decisions (finding the fastest path to the next waypoint). We propose macro-actions, depth limiting and hierarchical planning not as techniques specific to PTSP, but as promising directions for the application of MCTS and other discrete AI techniques to games and decision problems with large state spaces and/or real-time aspects in general.

The structure of this paper is as follows. Section II describes the PTSP and the associated competition. Section III presents our domain-specific approach to the higher-level decision problem of *route planning*, i.e. choosing the order in which to visit the waypoints. In Section IV we give our MCTS-based approach to the lower-level problem of *steering*, i.e. planning the sequence of actions to follow the route as quickly as possible. Section V presents results from the 2012 WCCI PTSP competition, justifying that our approach is very strong compared to other AI approaches and human players. Finally Section VI gives some concluding remarks and directions for future work, discussing how the ideas in this paper may apply to domains other than PTSP.

## II. The Physical Travelling Salesman Problem

The physical travelling salesman problem (PTSP) is an extension of the travelling salesman problem (TSP) which adds a physical environment. In the PTSP, a spaceship must visit all waypoints on a two-dimensional map in as short a time as possible. The ship moves according to a physical model based on Newtonian mechanics, updated in (simulated) real-time. This leads to a key distinction between the TSP and the PTSP: in the TSP the cost of travelling between two nodes is always the same; in the PTSP the cost of travelling between two waypoints depends on the ship's momentum and direction when it arrives at the first waypoint. The map also includes static obstacles. There is no penalty for colliding with obstacles, except that collisions are inelastic (the ship bounces off the obstacle and loses some, but not all, momentum).

The version of the PTSP being studied is that being used for the 2012 PTSP competition [2] held at WCCI 2012. The competition version of the PTSP had the following additional constraints:

- Entries are evaluated on maps which are not seen in advance but always contain 10 waypoints.
- Each entry is allowed to perform initial calculations for up to 1 second per map before the timer starts.
- An input must be made every 40ms and a waypoint must be visited at least every 1000 steps.

Control of the ship is discrete, and similar to the classic video game *Asteroids* [13]: the ship has a single thruster at the rear, so to accelerate in a particular direction it must first rotate (with fixed angular velocity) to face that direction. At each time step the agent must decide whether or not to fire the engine, and whether to rotate left, rotate right or not rotate. Thus $2 \times 3 = 6$ actions in total are available as inputs to the ship. Since there are 10 waypoints and one must be visited at least every 1000 simulation steps, the game lasts for up to 10000 steps. The maps used for the competition are not shown to competitors, but the software kit comes with 10 example maps which are also used for the human versus AI competition.

The PTSP is an interesting problem to study because it features some of the challenges presented by modern mainstream commercial video games, but has simple game mechanics. In particular it requires forward planning in a modelled physical environment, with decisions being made with high frequency. Additionally the PTSP competition takes place on unseen maps, which motivates the development of generic techniques that do not need expert knowledge or large amounts of upfront computation. Another advantage is that the PTSP competition allows human players to compete for high scores, which provides a useful benchmark.

## III. Route planning

We plan the order in which we will visit waypoints in an initial *route planning* stage, described in this section. Once the order of waypoints is established, we use an MCTS approach to steer along this route, discussed in Section IV. Note that the route specifies only the order of the waypoints, not the paths that must be taken between them. Since the PTSP agent has momentum and a maximum angular speed, the cost of the path depends not only on its length but also on the angles through which it turns.

The route planning process has two phases. First, a flood fill algorithm [14] is used to compute *distance maps*, which specify the distance of every waypoint from every other point on the map (Section III-A). These distance maps can be used to find shortest paths between waypoints; although the steering algorithm does not actually follow these paths, they can be used to estimate the angles of entrance and exit for each waypoint (Section III-B). The second phase uses these distances and angles to reduce the route finding problem to an instance of TSP, and uses the well-known multiple fragment [15] and 3-opt [16], [17] heuristics, suitably modified to bias towards routes without sharp turns, to find a solution (Section III-C).

### A. 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 steering algorithm (Section IV-C), it is necessary to find the distance between the ship and the next waypoint, again accounting for obstacles. Instead of computing these distances when needed, using A* pathfinding for example, we take advantage of the pre-processing time allowed by the rules of the PTSP competition and compute up-front the distance between every waypoint and every other non-obstacle point on the map.

Maps for the PTSP are represented as 2-dimensional arrays, where each cell is either an obstacle or open space. This

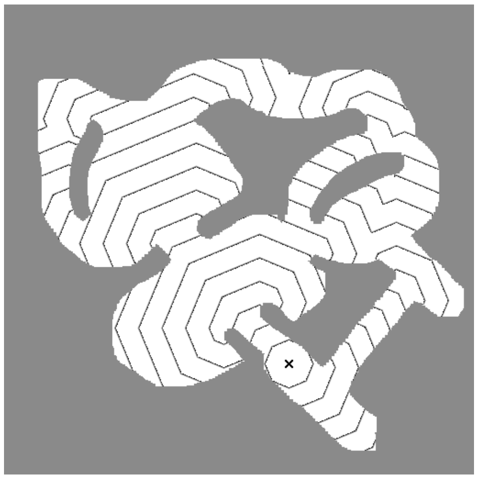

Fig. 1. An example of a distance map for the point marked $\times$. This is a contour map: lines show regions where the distance map value is a multiple of 25.

bitmap-like representation is particularly amenable to algorithms from computer graphics. Distance maps are computed using a modified scanline flood fill algorithm [14].

The distance map for waypoint $i$ is a 2-dimensional array $D_i$. After the distance map is computed, the entry $D_i[x, y]$ is the distance of point $(x, y)$ from waypoint $i$. Here "distance" is the minimal cost of a "king's move" path through the cells of the map, where a horizontal or vertical move has a cost of 1 and a diagonal move has a cost of $\sqrt{2}$.

The algorithm begins at waypoint $i$, and scans to the left and to the right until it encounters the first obstacle cell in each direction. Whilst scanning, each cell has its $D_i$ value updated according to the values of its neighbours, and unfilled non-obstacle cells immediately above and below the current scanline are enqueued to be used as starting points for subsequent scans. The algorithm terminates when the queue is empty, i.e. when all cells reachable from waypoint $i$ have been filled (assigned a value in $D_i$).

It is possible for a map to feature spaces or corridors too narrow to accommodate the ship. Before computing the distance maps, we surround every obstacle cell with a $+$-shaped region of obstacle cells, whose radius is equal to the ship's radius. A $+$-shaped rather than circular region is used purely for computational speed: for the purposes of removing narrow corridors, both are equally effective.

Figure 1 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. The algorithm could be modified to more closely approximate circular contours, but for our purposes this is a good tradeoff between speed and accuracy.

## B. Path directions

It is useful to estimate the angles at which the ship will enter and leave each waypoint on a particular route. This is

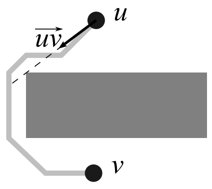

Fig. 2. Computing the path direction $\overrightarrow{uv}$. The thick grey line is the distance map traversal path, according to $v$'s distance map. The dotted line links $u$ with the first point on the path such that this line is obstructed. The vector $\overrightarrow{uv}$ is the unit vector in the direction of this line.

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 path from $u$ to $v$, as described above, until the first instance where the line between $u$ and the current point $p_i$ is obstructed. We then take $\overrightarrow{uv}$ to be the unit vector in the direction of $p_i - u$. This is illustrated in Figure 2. 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 [18] often used in pathfinding for video games.

## C. Planning the waypoint order

All of the maps used in the PTSP competition have exactly 10 waypoints. Solutions to 11-node min-weight Hamilton path problems which are close enough to optimal for our purposes can be found very quickly using the greedy multiple fragment heuristic [15] and 3-opt local improvement [16], [17].

3-opt normally seeks to minimise the sum of edge weights on the path. To account for the momentum of the agent, we instead seek 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

$$c(v_0, \ldots, v_n) = \sum_{i=1}^{n} d(v_{i-1}, v_i) + \beta_p \sum_{i=1}^{n} \frac{d(v_{i-1}, v_i)}{e(v_{i-1}, v_i)} + \beta_w \left( -\mathbf{u}_0 \cdot \overrightarrow{v_0 v_1} + \sum_{i=1}^{n-1} \overrightarrow{v_i v_{i-1}} \cdot \overrightarrow{v_i v_{i+1}} \right) \quad (1)$$

for constants $\beta_w$ and $\beta_p$. The first term is the sum of edge weights. The term modified by $\beta_p$ measures the directness of the path from one waypoint to the next, where a path is considered more indirect the more it deviates from a straight line in open space (i.e. the more the ratio of the path distance to the Euclidean distance increases). The term modified by $\beta_w$ 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).

## IV. Steering

This section describes the approach used to steer the ship along the route chosen by the methods in Section III, i.e. to plan and follow the paths from one waypoint to the next. (In line with video games terminology, "steering" here includes acceleration as well as rotation of the ship.)

We send an input to the ship once every 40ms, in order to steer the ship around obstacles and towards waypoints. If the choice of input on each time step is considered as a decision tree, then a tree search method can be used to determine the best input to make. On any map the number of steps taken will be between 1000 and 10000, so with 6 possible actions per step this means there are between $6^{1000}$ and $6^{10000}$ nodes in the tree. The problem is then to find the first terminal node in this tree (the shortest path to collect all 10 waypoints). The depth of this tree implies that an evaluation function for non-terminal states is needed.

The depth reached by any standard tree search approach within 40ms is unlikely to result in the ship moving significantly closer to any immediate goal (especially if the ship has low momentum). This has a big impact on the ability of tree search to plan far ahead in anticipation of obstacles and tight turns. Therefore we take an approach which drastically reduces the size of this tree by restricting the set of paths the steering controller can follow. This is done by building a tree of *macro-actions* where each macro-action takes multiple in-game steps to perform. Here we trade off the approximation of the decision process against the improvement in decision quality resulting from searching further forward in the decision tree.

The tree search algorithm we used is *Monte Carlo Tree Search* (MCTS) based on the UCT algorithm [19]. For this problem MCTS has a key advantage over other algorithms such as A* search and Minimax search in that it is anytime. This means that while a macro-action is being executed, several MCTS iterations can be performed during each 40ms decision step adding up to a sufficient number of iterations before the next macro-action is executed. This makes very efficient use of the computational time available which may have been difficult to achieve using alternative search methods. Additionally the asymmetric tree growth of MCTS means that more time will be spent searching branches of the tree that move the ship closer to the goal.

### A. Macro-actions

The steering problem for PTSP can be thought of as a directed graph, whose nodes are *states* and whose edges are *actions*. We define a *macro-action* as a sequence of actions $M = \langle a_1, \ldots, a_n \rangle$. *Executing a macro-action* corresponds to playing out its constituent actions in sequence. A decision tree of macro-actions can be built, whose nodes form a subset of the nodes of the underlying decision tree.

In 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 from state $s_0$, and $s_i$ is the state obtained by applying actions $a_1, \ldots, a_i$ in sequence from $s_0$, 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 increase the ability of tree search methods to perform forward planning. This can be achieved by coarsening the granularity of possible paths and preventing the ship from making small (sometimes meaningless) adjustments to speed and direction. The macro-actions we use in this paper consist simply of executing one of the six available actions (see Section II), say $a$, for a fixed number of time steps $T$: $M_a = \langle a, a, \ldots, a \rangle$, where $a$ appears $T$ times.

Earlier versions of our controller used more complex macro-actions, rotating to one of several pre-specified angles while thrusting or not thrusting. A problem arose relating to different actions taking different lengths of time to execute: since the evaluation function (Section IV-C) is implicitly a function of distance, MCTS tended simply to favour longer macro-actions over shorter ones. Having each depth of tree corresponding to a set amount of time and having MCTS roll out to a fixed depth means instead that MCTS tries to find short paths, by maximizing useful distance travelled in a fixed amount of time.

Since the game always takes between $[1000, 10000]$ steps, the game will take between $[\frac{1000}{T}, \frac{10000}{T}]$ macro-actions. The macro-action tree therefore contains between $[6^{\frac{1000}{T}}, 6^{\frac{10000}{T}}]$ nodes, which represents a hundreds of orders of magnitude reduction in the size of the problem to be solved (when $T \geq 2$). For example, suppose the length of the game is 2000 on average, and $T = 15$. The game tree contains $6^{2000} \approx 10^{1556}$ states, whereas the macro-action tree contains

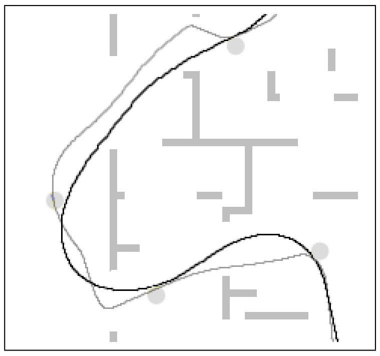

Fig. 3. Examples of the path followed by the MCTS controller. The light grey line has $T = 30$ corresponding to $90°$ turns. The black line has $T = 10$ corresponding to $30°$ turns.

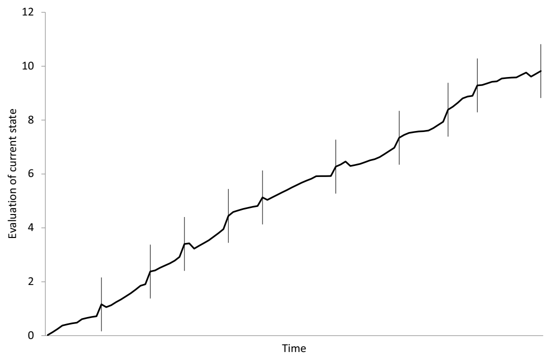

Fig. 4. A plot of the evaluation of the current state by Equation 2 against time, for an instance of PTSP. Vertical lines denote states where a waypoint was collected whilst executing the previous macro-action. Note the jump in evaluation score at these states.

$6^{\frac{2000}{15}} \approx 10^{103}$ 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 $9 \times 9$ 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, the MCTS algorithm 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 3 where the path with $90°$ turns is more jagged (and takes longer to follow) than with $30°$ turns. The price paid for path smoothness is tree size: to achieve the same look-ahead time as a $T = 30$ tree of depth $d$ and $N$ nodes, the $T = 10$ tree would need depth $3d$ and $O(N^3)$ nodes.

## B. MCTS

The MCTS algorithm works by iteratively building a search tree, adding one new node on each iteration. For an overview of MCTS methods see [6]. Each MCTS iteration consists of four distinct steps: *selection* (descending the portion of the tree already constructed), *expansion* (adding a node to the tree), *simulation* (playing out random moves to determine the outcome of the game), and *backpropagation* (updating the selected and expanded nodes according to the simulation result). One of the key features of the MCTS algorithm is that it expands the tree asymmetrically, balancing the exploitation of promising lines of play with the exploration of untried lines of play.

In the PTSP competition, the agent is allowed 40ms per decision. While executing the current macro-action, we use the CPU budget to continue searching the MCTS tree for the next macro-action. This ensures that by the time the next macro-action is required, the MCTS tree has accumulated several thousand iterations. To avoid timing out, we measure the maximum amount of time $m_T$ taken by an MCTS iteration so far, and stop searching after $38 - m_T$ milliseconds have elapsed.

Additionally each roll-out is performed up to a fixed depth, i.e. the total number of macro-actions performed during selection, expansion and simulation is constant across all MCTS iterations. This ensures that the evaluation function for nonterminal states is always applied to states at the same depth in the tree, and avoids looking ahead to regions of the tree unlikely to significantly affect the current decision.

## C. Evaluation Function

The evaluation of the ship being in a particular nonterminal state $s$ is calculated as

$$V(s) = \alpha_w s_w + \alpha_r(1 - s_r) + \alpha_s s_s \tag{2}$$

where $s_w$ is the number of waypoints that have been collected so far along the route recommended by the route-planning phase (Section III-C), $s_r$ is the distance of the ship from the next waypoint, normalised so that the distance from the previous waypoint to the next waypoint is 1 (Section III-A) and $s_s$ is the speed of the ship. The $\alpha$ values are weights to be tuned. Note that the evaluation explicitly does not reward the agent for collecting waypoints early (out of route order): otherwise the controller has a tendency to make detours to greedily collect waypoints, which generally turns out to be detrimental in the long term.

Choosing $\alpha_r < \alpha_w$ means that there is a jump in reward associated with collecting the next waypoint. See Figure 4. This encourages the controller to take short cuts to the next waypoint taking advantage of the physics of the game, which would not occur if the controller was exactly following the distance map traversal paths suggested by the route planning phase. Similarly the purpose of the speed feature is to encourage the controller to favour paths with more momentum 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):

$$V(s) = 10\alpha_w + \alpha_t (10000 - t) \quad (3)$$

where $t \leq 10000$ is the number of time steps taken to collect all the waypoints and $\alpha_t$ is a constant. This ensures that terminal states have an evaluation of at least $10\alpha_w$, which is higher than any non-terminal state, and that terminal states that are reached in less time have a higher evaluation that terminal states which took more time.

An initial implementation re-used the search tree statistics between macro-action decisions: upon executing a macro-action, the corresponding child of the root node was taken as the new root node. This seems like a good idea, since there should be useful information about future decisions in the existing tree and effectively provides extra MCTS iterations. However this surprisingly led to poor and erratic behaviour of the controller. One possible explanation is that the simulations are performed up to a fixed depth, so on subsequent decisions the reward values which initially exist in the tree correspond to a different depth and are not consistent with each other. Schemes for re-using previous search data are a subject of ongoing investigation.

## D. Parameters

Table I summarizes all the parameters for the controller with values that were tuned by hand through extensive trial-and-error tests. The biggest challenge of parameter tuning was to balance the macro-action length $T$ with the maximum MCTS search depth $d$. As illustrated in Figure 3 a lower value of $T$ results in smoother and potentially faster paths. However a lower value of $T$ (whilst keeping $d$ fixed) reduces the search horizon and limits the forward planning ability of the agent. Increasing $d$ increases the search horizon and improves the agents ability to perform forward planning but exponentially increases the size of the search tree. Consequently for higher values of $d$ fewer MCTS iterations will be performed in a given amount of time, while the larger tree needs more iterations to be searched effectively. This means that finer path granularity comes at the price of forward planning so a balance must be found.

| Parameter | Value | Description |
|---|---|---|
| $\beta_w$ | 50 | Penalty for sharp turns at waypoints (Equation 1) |
| $\beta_p$ | 100 | Penalty for indirect paths between waypoints (Equation 1) |
| $T$ | 15 | Length of macro-actions (Section IV-A) |
| $d$ | 8 | Maximum depth of MCTS rollout, in macro-actions (Section IV-B) |
| $C$ | 0.7 | Exploration constant for the UCB1 formula [19] |
| $\alpha_w$ | 1.0 | Weight applied to the number of waypoints visited (Equation 2) |
| $\alpha_r$ | 0.75 | Weight of the distance to the next waypoint (Equation 2) |
| $\alpha_s$ | 0.25 | Weight of the ship's speed (Equation 2) |
| $\alpha_t$ | 0.001 | Weight of time taken to reach a terminal state (Equation 3) |

TABLE I
LIST OF PARAMETERS FOR THE FINAL CONTROLLER. THESE ARE DIVIDED INTO THREE GROUPS: EVALUATION WEIGHTS FOR ROUTE PLANNING; MCTS PARAMETERS FOR STEERING; EVALUATION WEIGHTS FOR STEERING.

| Algorithm Version | Waypoints | Time |
|---|---|---|
| Nearest waypoint evaluation | 50 | 13249 |
| Distance maps with nearest waypoint ordering | 100 | 15888 |
| Distance maps with TSP waypoint ordering (distance only) | 100 | 14071 |
| Distance maps with TSP waypoint ordering | 100 | 13565 |

TABLE II
THE IMPROVEMENTS GAINED BY EACH ASPECT OF OUR APPROACH ACROSS THE 10 PTSP COMPETITION STARTER KIT MAPS

## V. PERFORMANCE

Table II summarises the improvement due to each aspect of our approach. The table compares four controllers:

- Nearest waypoint evaluation: The evaluation used by MCTS is as Equations 2 and 3, but $s_r$ is the Euclidean distance (ignoring obstacles) between the ship and the nearest waypoint.
- Distance maps with nearest waypoint ordering: The evaluation used by MCTS is as Equation 2, and $s_r$ is obtained from the distance maps as with the final controller, but the next waypoint is the one that is nearest to the ship by Euclidean distance.
- Distance maps with TSP waypoint ordering (distance only): The ordering of the waypoints is computed as described in Section III-C but only the distances (derived from distance maps) are used in Equation 1 ($\beta_w = \beta_p = 0$).
- Distance maps with TSP waypoint ordering: The complete algorithm used for the PTSP competition.

In each case, the other details of the algorithm (including all parameter values) are as in the final controller.

The controller with nearest waypoint evaluation was unable to find all 100 waypoints, due to having no way to find paths around obstacles. All other versions successfully visited all 100 waypoints. The biggest improvement comes from planning the order in which waypoints will be visited. The addition of direction and directness heuristics to route planning significantly improves performance on three of the maps, by approximately 200 time steps in each case. On the other seven maps these heuristics do not change the waypoint ordering and so the time taken is unchanged.

Table III presents the scores achieved by our agent on the 10 maps issued with the PTSP competition starter kit, taken from the combined Human vs Bot results from the WCCI 2012 PTSP competition [2]. The scores recorded for each agent and player are the best achieved over many trials. The competition ran from 1 March 2012 to 28th May 2012. The uploaded AI submissions get 5 attempts per map each time a new version is uploaded, with the fastest time recorded; human players can make an unlimited number of attempts through

| Map | Our Score | Our Rank | Best Score | Next Best Score |
|---|---|---|---|---|
| 1 | 1298 | 3 | 1286 | - |
| 2 | 1043 | 4 | 973 | - |
| 3 | 1132 | 2 | 1117 | - |
| 4 | 1312 | 2 | 1311 | - |
| 5 | 1641 | 3 | 1580 | - |
| 6 | 1367 | 1 | - | 1381 |
| 7 | 1188 | 2 | 1181 | - |
| 8 | 1464 | 2 | 1451 | - |
| 9 | 1721 | 1 | - | 1785 |
| 10 | 1156 | 7 | 1102 | - |

TABLE III
A SUMMARY OF THE SCORES ACHIEVED BY OUR AGENT IN THE 2012 WCCI HUMAN VS. BOT PTSP COMPETITION. HERE THE "SCORE" IS THE NUMBER OF TIME STEPS TO COLLECT ALL TEN WAYPOINTS ON THE MAP.

the website. Listed in Table III is the best score achieved by our agent and our rank on each map. For maps where our agent did not achieve the best score, the best score achieved by another agent/human visitor is listed. For maps where our agent achieved the best score, the score of the agent/human in second place is listed. In every case in this table, the best score or next best score was achieved by a human player. Our agent was the best AI on seven of the 10 maps, and second best on the other three.

Our agent has very competitive performance compared to human players, being amongst the top 3 players on 8 out of the 10 maps and being behind by only a small margin on other maps. The map with the worst result is map 10, which is fully open with no obstacles. Here the techniques applied for planning routes and avoiding obstacles make little impact on the performance of the algorithm. On the other hand, our agent achieves the winning score on map 9, a maze-like map with sharp turns and narrow corridors. This demonstrates the effectiveness of our route planning and steering approaches for more complex problems. Overall our entry placed second in the Human vs. Bot competition, which was won by a human player. Competition was close however: the winning player's average time per map was only 15.5 time steps faster than our agent's average time.

Table IV lists the competition results for the 2012 WCCI PTSP competition, which had 31 submitted AI controllers. Only results for controllers which scored points in the competition are listed. There were 20 hidden maps and submissions were ranked for each map, then the final scores were derived by awarding submissions points based on their rank on each map. Our agent (username "Purofvio") was the highest ranked submission on 18 out of 20 maps, and second highest on the remaining two maps. This result is a convincing victory, easily outperforming all other AI entries.

## VI. CONCLUSION AND FUTURE WORK

This paper presented an MCTS-based controller for PTSP. In the WCCI 2012 competition, our approach beat all other AI players (Table IV) and was highly competitive with the best human players (Table III). Our approach has three key features:

| Rank | Username | Affiliation | Points Awarded |
|---|---|---|---|
| 1 | Purofvio | University of Bradford | 196 |
| 2 | st3f1 | University of Le Havre | 124 |
| 3 | shavlir | none | 121 |
| 4 | ICE_DE | Ritsumeikan University | 88 |
| 5 | hiten_sonpal | AI Hobbyist | 84 |
| 6 | melsonator | None | 64 |
| 7 | IMarvinTPA | IMarvinTPA | 47 |
| 8 | BBoy_Alin | Essex uni | 19 |
| 9 | axiom | RPI | 11 |
| 10 | darklink259 | RPI | 6 |
| 11 | Arkanis1991 | UC3M | 5 |
| 12 | gongj2 | RPI | 4 |
| 13 | eoinomurchu | University College Dublin | 3 |
| 14 | Rocket | None | 3 |
| 15 | aguc3m | UC3M | 2 |
| 16 | nbgray | None | 2 |
| 17 | kerlak | UC3M | 1 |

TABLE IV
COMPETITION RESULTS FOR THE WCCI 2012 PTSP COMPETITION.

1) The use of macro-actions to reduce the size of the game tree by hundreds of orders of magnitude, and hence allow search to look forward much further in time;
2) A hierarchical approach, in which a higher-level AI reduces the problem to a sequence of sub-goals and a lower-level AI achieves these sub-goals in sequence;
3) Using depth-limited rollouts within MCTS, with an evaluation function representing achievement of high-level sub-goals at nonterminal states.

One important point about the hierarchical approach is that the lower-level AI is not simply trying to achieve the current sub-goal. The evaluation function is arranged to give a gradual increase when progressing towards the current sub-goal and a discontinuous jump upon achieving that sub-goal, as illustrated in Figure 4. Thus the MCTS steering controller favours lines of play that lead to achieving the current sub-goal (i.e. passing through the next waypoint on the route), but amongst those it prefers lines of play that put it in a more favourable position to achieve the sub-goal after that.

The introduction of depth limiting and heuristic evaluation is a major factor in the success of this approach, despite the fact that it negates features of MCTS that are usually quoted as its advantages over minimax search (the lack of a need for a heuristic and, to an extent, the asymmetric depth of the tree). However when MCTS is used to achieve short-term goals in a hierarchical system, depth limiting makes sense. As shown in Section V by the poor performance of our steering controller coupled with greedy waypoint selection, the hierarchical approach encompassing both route planning and steering is needed to create competitive PTSP solutions.

Our heuristics, though domain dependent, do not rely on what we would term "expert" domain knowledge. The heuristic used in route planning relies only on the very basic intuition that sharp turns should be avoided. The heuristic used in steering is based on the almost trivial observation that moving towards the next waypoint is beneficial, with the more complex notions of which waypoint should be next and what it means to move closer to it being determined by our route planner AI.

Our AI for route planning is heavily domain dependent, relying not on general-purpose game tree search but on special-purpose algorithms and heuristics. Other approaches were tried which used MCTS for route planning. These were effective but took much longer to execute than the approach given in Section III. They did however have a feature that may be useful when the hierarchical approach is applied to other domains: the steering controller could begin to follow a partial route while the route planner was still working to refine it.

These techniques are ripe for application to other domains, and this is a subject of current and future work. Possible domains include car racing, first-person shooter games, strategy games, and other video games which simulate movement. As an anytime algorithm, MCTS has great potential for use in real-time environments since it can make efficient use of whatever computational resources are available.

In real-time domains the ability to make a different decision on every time step can be unnecessary, and results in intractably large game trees. Simple macro-actions consisting of repeating the same action for a fixed number of time steps are a simple yet effective means of drastically reducing the size of this tree. More complex macro-actions, such as repeating an action until some condition is met or performing a sequence of possibly different actions, may also be useful in some domains. Another interesting direction for future investigation would be to vary the macro-action length between levels in the tree, allowing for more fine-grained control in the short term while retaining the advantages of coarse-graining in the long term.

In real-time games, and other domains such as the class of complex multiplayer strategy-heavy turn-based games known as “Eurogames” (of which Settlers of Catan [20] is a well-known example), a game can consist of hundreds if not thousands of turns. This can make searching the game tree for a winning terminal state almost impossible. However as humans, we tend to break these games down into hierarchies: the main goal (“win the game”) is decomposed into several sub-goals (“capture territory $X$”, “maximise the amount of resource $Y$”), which are decomposed in turn until we reach the level of individual actions. A hierarchical approach to such games is attractive, and allows the flexibility to tailor the AI technique used at each level in the hierarchy to the nature of that particular decision problem. Hierarchical approaches to planning have been considered in [21] for example.

In our PTSP controller, the higher-level route planner feeds into the lower-level steering controller by means of an evaluation function. There is no feedback from lower to higher levels. In other domains it may be useful to introduce such feedback, with the lower-level planner feeding information about the probability or difficulty of achieving certain sub-goals to the higher-level planner, which could use this new information to refine its plan.

MCTS tends to be tactically strong but strategically weak. Arguably this is a limitation of game tree search, which by its nature focusses on the next action to play rather than the bigger picture. Macro-actions and hierarchical planning allows MCTS to take a more strategic approach, by reducing strategic decisions to tactical decisions on an abstracted higher-level game. This is a promising approach for many games currently considered difficult for search-based AI.

## Acknowledgements

We thank the organisers of the PTSP competition [2], with particular thanks to Diego Perez for his assistance in working with the software framework. We also thank the reviewers for their insightful comments.

## References

[1] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, Eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley and Sons, 1985.

[2] D. Perez, D. Robles, and P. Rohlfshagen, “The Physical Travelling Salesman Problem Competition Website,” 2012. [Online]. Available: http://ptsp-game.net/

[3] G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck, “Monte-Carlo Tree Search: A New Framework for Game AI,” in *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, Stanford Univ., California, 2008, pp. 216–217.

[4] C.-S. Lee, M. Müller, and O. Teytaud, “Guest Editorial: Special Issue on Monte Carlo Techniques and Computer Go,” *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 225–228, Dec. 2010.

[5] A. Rimmel, O. Teytaud, C.-S. Lee, S.-J. Yen, M.-H. Wang, and S.-R. Tsai, “Current Frontiers in Computer Go,” *IEEE Trans. Comp. Intell. AI Games*, vol. 2, no. 4, pp. 229–238, 2010.

[6] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A Survey of Monte Carlo Tree Search Methods,” *IEEE Trans. Comp. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, 2012.

[7] S. Samothrakis, D. Robles, and S. M. Lucas, “A UCT Agent for Tron: Initial Investigations,” in *Proc. IEEE Symp. Comput. Intell. Games*, Dublin, Ireland, 2010, pp. 365–371.

[8] N. G. P. Den Teuling, “Monte-Carlo Tree Search for the Simultaneous Move Game Tron,” B.Sc. Thesis, Univ. Maastricht, Netherlands, 2011.

[9] S. Samothrakis, D. Robles, and S. M. Lucas, “Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man,” *IEEE Trans. Comp. Intell. AI Games*, vol. 3, no. 2, pp. 142–154, 2011.

[10] K. Q. Nguyen, T. Miyama, A. Yamada, T. Ashida, and R. Thawonmas, “ICE gUCT,” Int. Comput. Entertain. Lab., Ritsumeikan Univ., Tech. Rep., 2011.

[11] N. Ikehata and T. Ito, “Monte-Carlo Tree Search in Ms. Pac-Man,” in *Proc. IEEE Conf. Comput. Intell. Games*, Seoul, South Korea, 2011, pp. 39–46.

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

[13] Atari, “Asteroids.” [Online]. Available: http://www.atari.com/asteroids

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

[15] J. L. Bentley, “Experiments on Traveling Salesman Heuristics,” in *Proc. 1st Annu. ACM-SIAM Symp. Disc. Alg.*, 1990, pp. 91–99.

[16] S. Lin, “Computer solutions of the traveling salesman problem,” *Bell Syst. Tech. J.*, vol. 44, pp. 2245–2269, 1965.

[17] D. S. Johnson and L. A. McGeoch, “The Traveling Salesman Problem: A Case Study in Local Optimization,” in *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra, Eds. John Wiley and Sons, 1997, pp. 215–310.

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

[19] L. Kocsis and C. Szepesvári, “Bandit based Monte-Carlo Planning,” in *Euro. Conf. Mach. Learn.*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Germany: Springer, 2006, pp. 282–293.

[20] BoardGameGeek, “Settlers of Catan.” [Online]. Available: http://boardgamegeek.com/boardgame/13/the-settlers-of-catan

[21] A. Botea, Martin Müller, and J. Schaeffer, “Near optimal hierarchical path-finding,” *Journal of Game Development*, vol. 1, pp. 7–28, 2004.