Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games

Diego Perez
University of Essex
Colchester CO4 3SQ
United Kingdom
dperez@essex.ac.uk

Spyridon Samothrakis
University of Essex
Colchester CO4 3SQ
United Kingdom
ssamot@essex.ac.uk

Simon M. Lucas
University of Essex
Colchester CO4 3SQ
United Kingdom
sml@essex.ac.uk

Philipp Rohlfshagen
University of Essex
Colchester CO4 3SQ
United Kingdom
prohlf@essex.ac.uk

ABSTRACT

In real-time games, agents have limited time to respond to environmental cues. This requires either a policy defined up-front or, if one has access to a generative model, a very efficient rolling horizon search. In this paper, different search techniques are compared in a simple, yet interesting, real-time game known as the Physical Travelling Salesman Problem (PTSP). We introduce a rolling horizon version of a simple evolutionary algorithm that handles macro-actions and compare it against Monte Carlo Tree Search (MCTS), an approach known to perform well in practice, as well as random search. The experimental setup employs a variety of settings for both the action space of the agent as well as the algorithms used. We show that MCTS is able to handle very fine-grained searches whereas evolution performs better as we move to coarser-grained actions; the choice of algorithm becomes irrelevant if the actions are even more coarse-grained. We conclude that evolutionary algorithms can be a viable and competitive alternative to MCTS.

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence—*learning*

Keywords

ai, games, real time, evolutionary algorithms, mcts

1. INTRODUCTION

Traditionally, research in Game AI has focused on 2-player perfect information games where the time allowed to make a move is measured in seconds or minutes. Lately, research has started to focus on more complex domains such as those where the whole state of the game is unobserved by the players (as in many card games) or is affected by random events (e.g. in stochastic games like Backgammon). Likewise, real-time constraints have attracted increasing attention, with particular emphasis on video games, where moves need to be made every number of milliseconds. In this paper we will concentrate on the real-time aspect of a game, which requires a search algorithm to converge very fast without much overhead. It is therefore worthwhile to investigate how search methods are able to explore the search space of those games characterised by limited decision time.

The benchmark (and the experimental testbed) used in this paper is the Physical Travelling Salesman Problem (or PTSP), a deterministic single player game that consists of a collection of several waypoints scattered around a map full of obstacles. The challenges here include finding the best sequence of waypoints, as well as the challenge of navigating the player to visit each waypoint in sequence in as little time as possible. This paper focuses in the navigation aspect of the problem. As a real-time game, PTSP provides a small time budget to decide the next move of the player. Additionally, the game allows the player to run simulations in order to foresee the outcome of applying any of the available actions. However, the final state of the game is usually never reached, as the number of moves required to do so by far exceeds the time available to execute them.

As a single-player game, the PTSP allows players that have a model of the game to predict the rewards of the actions taken without any ambiguity. This feature allows the controllers to create precise macro-actions (or repetitions of the same single actions) to reach further states when evaluating the next move to make.

The PTSP has been selected because it holds certain similarities with some modern video-games, at a simplified level. For instance, real-time games need an action to be supplied within a few milliseconds, providing a very limited budget to plan the next move. Furthermore, as in the PTSP, the duration of the game is long enough so the agent is not able to foresee all the outcomes from the moves made. Finally, the fact that the PTSP has featured in international conferences as a competition (see Section 3) allows for a direct compar-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.
Copyright 2013 ACM 978-1-4503-1963-8/13/07 ...$15.00.

ison between the approaches shown here and the ones from the contest.

Four different algorithms are presented in this paper: two versions of a Rolling Horizon Genetic Algorithm (GA and GAC), Rolling Horizon Random Search (RS) and Monte Carlo Tree Search (MCTS). Each one of these algorithms follows a similar scheme in order to solve the problem: first, the order of waypoints to visit is obtained by a common TSP solver. Then, a macro-action handler performs the desired moves while it requests future actions from the appropriate algorithm. Each algorithm performs the search differently, but uses the same value function that evaluates a given state of the game. Therefore, the focus of this paper is on the ability of the algorithm to explore the search space, using different macro-action lengths.

This paper is organised as follows. Section 2 discusses tree search and evolutionary algorithms in real-time games. Section 3 describes the PTSP and Section 4 introduces the algorithms and controllers used to play the game. Section 5 details the experimental setup and discusses the results obtained. The paper is concluded in Section 6, where some possible extensions of this work are discussed.

2. LITERATURE REVIEW

2.1 Monte Carlo Tree Search

MCTS is a tree search technique that has been widely used in board games, particularly in the game of Go. Go is a territorial board game played in a 19×19 square grid, in which the two players aim to surround their opponent's stones. The game of Go is considered the drosophila of Game AI, and MCTS players have achieved professional level play in the reduced version of the game (9×9 board size) [8].

MCTS is considered to be *anytime*: the algorithm is able to provide a valid solution (next move in the game) at any moment in time, independently from the number of iterations that have been performed (although more iterations generally produce better results). In contrast, other algorithms (such A*) usually require to have finished to provide an initial action to execute. For this reason, MCTS is a suitable method for real-time domains, and it has been employed extensively in the literature.

A popular real-time game that has been employed as a benchmark for using MCTS is Ms. Pac Man. In this game, the user controls the Pac Man agent with the objective of clearing the maze and eating all the pills without being captured by the ghosts. This game, similar to the PTSP, is open ended (i.e. the end game state is usually not reached within the simulations). Robles et al [16] include hand-coded heuristics to guide the simulations towards more promising portions of the search space. Other authors also include domain knowledge from the game to bias the search in MCTS, such as [18, 7].

Regarding single player games, MCTS has been used in several puzzles. Examples are SameGame [9], where the player aims to destroy contiguous items of the same colour distributed in a rectangular grid, or Morpion Solitaire [4], a connection puzzle where the objective is to join vertices of a graph with lines that contain at least five nodes. PTSP itself has been addressed with MCTS techniques, as Perez et al. [11] did to show the importance of adding heuristics to the Monte Carlo (MC) simulations in real-time domains, or Powley et al. [14], who describe the MCTS algorithm that won them the first edition of the PTSP competition.

An extensive survey of Monte Carlo Tree Search methods has been written by Browne et al. [2], where the algorithm, variations and applications are described. It is worth noting that Monte Carlo Tree Search, when combined with UCB1 (see later in this paper) reaches asymptotically logarithmic regret [3]. One can intuitively think of regret as "opportunity loss".

2.2 Rolling Horizon Evolutionary Algorithms

When it comes to planning (or control), evolutionary algorithms are mostly used as follows: an evolutionary algorithm is used in conjunction with an off-line simulator in order to train a controller, for example as in [6]. The controller is then used in the real problem. This approach is not just used for control problems, but has been popular in fields such as Artificial Life (e.g. [1]). This paper proposes a different approach. Evolution is used in the same manner as MCTS uses roll-outs and the generative model (i.e. a simulator). Thus, an agent will evolve a plan in an imaginary model for some milliseconds, acts on the (real) world by performing the first action of its plan and then evolves a new plan repeatedly (again in a simulation based manner) until the game is over. This type of behaviour is called "Rolling Horizon", "Receding Horizon" or "Model Predictive" control or planning. There have been some efforts in the past to use such approaches (e.g. [17, 10], and this paper presents two versions of a genetic algorithm in this setting. The term Rolling Horizon comes from the fact that planning happens until a limited look-ahead in the game (and thus we have to keep on replanning at every time step). Contrary to MCTS, we are currently unsure about the regret bounds of genetic algorithms in the general case [15].

3. THE PHYSICAL TRAVELLING SALESMAN PROBLEM

The Physical Travelling Salesman Problem (PTSP) is a modification of the well known optimisation problem Travelling Salesman Problem (or TSP).[1] In the PTSP, the player controls a ship (or agent) that must visit a series of waypoints scattered around a continuous 2-D maze full of obstacles. The score of the game is determined by the number of waypoints visited and the time taken to do so. Figure 1 shows one of the maps of this game.

The PTSP is a real-time game. For every cycle, the ship must supply an action to execute in no more than $40ms$. Additionally, there is a maximum time, depending on the number of waypoints in the map, that cannot be overspent when visiting the next waypoint. There are six different actions that can be provided to steer the ship. These actions are discrete and are divided into two different inputs: acceleration (which can be *on* or *off*) and steering (which can be *left*, *right* or *straight*). The action applied modifies the position, velocity and orientation of the ship.

The PTSP featured in two competitions in 2012 which

[1]TSP consists of an agent who must visit all n cities once via the shortest possible route, starting and ending at the same location.

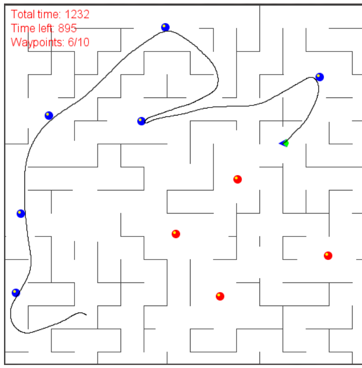

Figure 1: Sample PTSP Map.

took place at international IEEE conferences: the World Congress On Computational Intelligence (WCCI) and the Conference on Computational Intelligence and Games (CIG). The interested reader can found more information about the competition (game, rules and framework), as well as background for TSP solvers, navigation algorithms and similar real-time game competitions at [13]. The winner of both competitions was a Monte Carlo Tree Search approach based on macro-actions and a physics-based TSP route planner [14].

4. SOLVING THE PTSP

An obvious way to tackle the PTSP is by approaching the two sub-problems it constitutes: the order of waypoints to follow and the procedure to drive the ship to each of them. In other words, it is necessary to tackle both short-term and long-term planning tasks, whilst taking into account the interdependencies between the two (i.e. the physics of the game).

The focus of this paper is on the driving procedure. The order of waypoints followed is the same for the different algorithms presented in this study. This route is calculated at the beginning of the game, during initialization, and it remains available to the controller for the whole game. The calculated route takes the physics of the game into account: each given route has an associated cost assigned by a procedure that decomposes the route into straight-line obstacle free segments. The time taken to complete each one of these segments is estimated using the physics of the game and keeping the speed at the end of the segments. A penalization factor is applied at the end of each segment, related to the angle between the current and next segment in the route. This factor reduces the speed of the agent after each segment, in order to approximate the effects of turns in the trajectory of the ship. It has been determined [12] that those routes involving fewer changes of direction, even when the distance travelled is higher, usually provide better results.

4.1 Single versus Macro Actions

As stated in Section 3, there are six different actions that can be applied at each time step. Each action must be decided within the 40ms of time allowed for the controller to choose a move. A good quality solution for a PTSP map with 10 waypoints requires between 1000 (in the maps with less obstacles) and 2000 (in more complex ones, as in the one shown in Figure 1) time steps or action decisions. This creates a search space of $(1000, 2000) \rightarrow (6^{1000}, 6^{2000})$, which obviously makes it impossible to calculate the optimal route within the time limitations given.

A reasonable reduction of this search space can be achieved by limiting how far ahead in the game the algorithms can look. As described before, the order of waypoints is calculated during the initialization process. The controller has information about how many and which waypoints have been already visited, and it knows which ones are the next N waypoints to be collected. If only the next 2 waypoints are accounted, and according to the game length estimates, the search space size is reduced to $(6^{2\times100}, 6^{2\times200})$ (assuming that the ship usually takes 100 to 200 actions per waypoint).

More reduction can be achieved by introducing the concept of *macro-actions*. In this paper a macro-action is just the repetition of the same action for L consecutive time steps. The impact of changing a single action in a given trajectory is not necessarily big, so high quality solutions may still be found by not using such fine-grained precision. For instance, choosing a macro-action *length* (defined as the number of repetitions per macro-action) of 10, reduces the search space size to $(6^{20}, 6^{40})$. Different values of L are used during the experiments described in this paper in order to show how this parameter affects the performance of each one of the algorithms described.

Macro-actions have a second and important advantage: they allow longer evaluations to execute before taking a decision (through several game steps). In a single action scenario, each algorithm has only 40ms to decide the next action to execute. However, in the case of a macro-action that is composed by L single actions, each macro-action has $L \times 40$ms to decide the move, as the previous macro-action needs L time steps to be completed (with the exception of the very first move, where there is no previous action). Algorithm 1 defines a handler for macro-actions.

Algorithm 1 Algorithm to handle macro-actions.

```
 1: function GetAction(GameState : gs)
 2:     if IsGameFirstAction(gs) then
 3:         actionToRun ← DecideMacroAction(gs)
 4:     else
 5:         for i = 0 → remainingActions do
 6:             gs.Advance(actionToRun)
 7:         if remainingActions > 0 then
 8:             if resetAlgorithm then
 9:                 Algorithm.Reset(gs)
10:                 resetAlgorithm ← false
11:             Algorithm.NextMove(gs)
12:         else                          ▷ remainingActions is 0
13:             actionToRun ← DecideMacroAction(gs)
14:     remainingActions = (remainingActions − 1)
15:     return actionToRun
16:
17: function DecideMacroAction(GameState : gs)
18:     actionToRun ← Algorithm.NextMove(gs)
19:     remainingActions ← L
20:     resetAlgorithm ← true
21:     return actionToRun
```

The algorithm works as follows: the function *GetAction*, called every game step, must return a (single) action to ex-

ecute in the current step. When the function is called for the first time (line 3), or the number of remaining actions is back to 0 (line 13), the function $DecideMacroAction$ is called. This function (lines 17 to 21) executes the decision algorithm (which could be tree search, evolution or random search), that is in charge of returning the move to make, and sets a counter ($remainingActions$) to the length of the macro-action (L). Before returning the action to run, it indicates that the next cycle in the algorithm needs to be reset.

In cases where the current step is not the first action of the game, the state is advanced until the end of the macro-action that is currently being executed (line 6). This way, the decision algorithm can start the simulation from the moment in time when the current macro-action will have finished. The algorithm is then reset (if it was previously indicated to do so by $DecideMacroAction$) and finally evaluates the current state to take the next move (line 11).

Note that in this case the value returned by $NextMove$ is ignored: the action to be executed is *actionToRun*, decided in a different time step. The objective of this call is to keep building on the knowledge the algorithm is gathering in order to decide the next macro-action (which effectively happens in $DecideMacroAction$) while the ship makes the move from the `previous` macro-action decision. Therefore, $Algorithm.NextMove$ is called L times, but only in the last one is a real decision made.

It is important to note that the algorithms determine a best solution of N macro-actions with L single actions per macro-action. However, only the `first` macro-action of the best solution is finally handled and executed by the controller. Once the best individual has been determined after L iterations, the first macro-action is used and the rest are discarded, as the algorithm is reset at that point. The decision algorithm is in charge of keeping its internal state from one call to the next, replying within the allowed time budget and resetting itself when $Algorithm.Reset$ is called.

4.2 Score function

Even the usage of macro-actions does not ensure that any succession of N actions reaches the next two waypoints in the (pre-planned) route: it is unlikely that a random sequence of actions visits even one of them. In fact, the vast majority of random trajectories will not visit any waypoints at all. It is thus necessary to improve the granularity of the rewards by defining a function that is able to distinguish the quality of the different trajectories, by scoring the game in a given state. This function is defined independently from the algorithm that is going to use it, so the research is centred on how the search space is explored by the different algorithms.

The score function takes into account the following parameters: distance and state (visited/unvisited) of the next two waypoints in the route, time spent since the beginning of the game and collisions in the current step. The final score is the addition of four different values. The first one is given by the *distance points*, which is defined as follows: being d_w the distance to the waypoint w, the reward for distance r_{dw} is set to $10000 - d_w$. If the first waypoint is not visited, it is set to r_{d1}. However, if the first waypoint is visited, this is set to r_{d1} plus a reward value (10000). The other values for the score function are given by the waypoints visited (number of visited waypoints, out of the next two, multiplied by a reward factor, 1000), time spent (set to 10000 minus the time spent during the game) and collisions (-100 if a collision is happening in this step).

4.3 Genetic Algorithm

The two versions of the Genetic Algorithm (GA) employed in this study are simple evolutionary algorithms, where each individual encodes a group of macro-actions using integers. Each one of the possible six actions is represented with a number from 0 to 5. Note that each one of these genes represents a macro-action: the action denoted by the gene is repeated for L steps in the game where L is defined in the experimental setup.

All individuals in the population have a fixed length, as defined in the experimental setup phase. As the algorithm is meant to run online (i.e. while the game is actually being played), a small population size of 10 individuals is used in order to be able to evolve over a large number of generations. Elitism is used to automatically promote the best 2 individuals of each generation to the next. Evolution is stopped when $(40ms - \epsilon)$ is reached. ϵ is introduced in order not to use more time than allowed to provide an action.

Individuals are initialized uniformly at random such that each gene can take any value between 0 and 5. This initialization happens many times during each game: in the first step of the game and every L moves, as described in Section 4.1. Mutation is used with different rates: 0.2, 0.5 and 0.8. Each time a gene is mutated, only one of the two inputs involved in the action is randomly chosen to be changed. If acceleration is changed, the value is flipped from *On* to *Off*, or vice versa. If steering is modified, *left* and *right* mutate to *straight*, while *straight* can mutate to *right* or *left* at 50% chance. This procedure was designed in order to transit smoothly between actions. Tournaments of size 3 are used to select individuals for uniform crossover.

Finally, the evaluation of an individual is performed by the following mechanism: given an individual with N macro-actions and L single actions per macro-action, the game is advanced, using the forward model mentioned in Section 3, applying the $N \times L$ actions defined in the individual. Then, the resultant game state is evaluated with the score function defined in Section 4.2.

Two different variants are employed in this study. The first one (referred to as GAC), implements all features described above. The second one (herein simply referred to as GA) does not employ tournament selection nor crossover.

4.4 Monte Carlo Tree Search

MCTS is a simulation-based algorithm that builds a tree in memory. Each node keeps track of statistics relating to how often a move is played from a state ($N(s,a)$), how many times each move is played from a state ($N(s)$) and the value of the average reward ($Q(s,a)$). The algorithm presented here uses the Upper Confidence Bound (UCB1; see Equation 1) equation at each decision node to balance between exploration and exploitation. This balance is achieved by setting the value of C, which provides the algorithm with more or less exploration, at the expense of exploitation. A commonly used value is $\sqrt{2}$, as it balances both facets of the search when the rewards are normalized between 0 and 1.

$$a^* = \operatorname*{arg\,max}_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \quad (1)$$

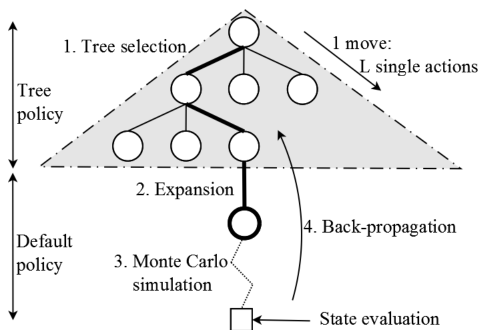

Figure 2: MCTS algorithm steps.

The MCTS algorithm is executed in several steps [5]: at the very beginning of the algorithm, the tree is composed of only the root node, which is a representation of the current state of the game. During the *selection* step, the tree is navigated from the root until a maximum depth has been reached. If a node has as many children as the available number of actions from that position, UCB1 is used as tree policy. Otherwise, a new node is added as a child of the current one (*expansion* phase) and the *simulation* phase starts. At this point, MCTS executes a Monte Carlo simulation (or roll-out; default policy) from the expanded node until the pre-defined depth, where the state of the game is evaluated. Finally, during the *back-propagation* step, the statistics $N(s)$, $N(s,a)$ and $Q(s,a)$ are updated for each node traversed, using the reward obtained in the evaluation of the state. These steps are executed in a loop until a termination criteria is met (in the case presented here, when the time budget is consumed). Figure 2 summarizes the steps of the algorithm.

In the problem presented in this paper, each of the actions in the tree is a macro-action. This means that, when navigating through the tree, L single actions must be performed to move from one state to the next, defined by the action picked. In most games, the depth of the tree is limited by the number of moves required to reach an end game state. In the case of the PTSP, the end of the game is usually not reached, so the maximum depth is determined by the number and length of the macro-actions, defined differently for each experiment, bringing it closer to rolling horizon search then normal MCTS.

The tree is preserved from one game step to the next (i.e. during L moves) and is discarded every time the algorithm is reset. When the controller asks for a move, the action chosen is the one that corresponds to the child of the root with the maximum expected value $Q(s,a)$.

4.5 Random Search

Random Search (RS) is a simple algorithm that creates random solutions of a specified length. During the allocated time, new individuals are created uniformly at random and evaluated, keeping the best solution found so far. As in the other solvers, the best solution is forgotten every time the algorithm is told to be reset. The motivation behind including RS in this research is the good results obtained by the GA with high mutation rates. Thus, it is important to investigate how the results given by RS compare to those.

5. EXPERIMENTATION

5.1 Experimental Setup

The experiments presented in this paper have been carried out on 10 different maps, running 20 matches per maze. Hence, each configuration has been executed during 200 games, in order to get reliable results.[2] The maps are those distributed within the framework of the PTSP competition.[3]

Five different configurations are used for each algorithm, varying the number of macro-actions (N) and the number of single actions per macro-action (L). These configurations, indicated by the pairs (N,L), are as follows:

$$\{(50,1),(24,5),(12,10),(8,15),(6,20)\}$$

The last four configurations share the property of foreseeing the same distance into the future: $N \times L = 120$. The only exception is the first configuration, where $L = 1$, this is considered as a special case as no macro-actions are used.

A key aspect for interpreting the results is to understand how games are evaluated. If two single games are to be compared, following the rules of PTSP, it is simple to determine which game is better: the solution that visits more waypoints is the best. If both solutions visit the same number of waypoints, the one that does so in the minimum number of time steps wins.

However, when several matches are played in the same map, determining the winner is not that simple. One initial approach is to calculate the averages of waypoints visited and time spent, following the same criteria used in the single game scenario. The problem with this approach is that the difference between two solutions, taken as the average of waypoints visited, can be quite small (less than one waypoint), while the difference in time steps can be large. For instance, imagine a scenario where two solvers (A and B) obtain an average of waypoints (w_i) of $w_a = 9.51$ and $w_b = 9.47$, and an average of time steps (t_i) of $t_a = 1650$ and $t_b = 1100$. The traditional comparison would say that A wins (because $w_a > w_b$), but actually B is much faster than A ($t_b \ll t_a$) with a very small waypoint difference. Intuitively, B seems to be doing a better job.

A possible solution to this problem is to calculate the ratio $r_i = t_i/w_i$ that approximates the time taken to visit a single waypoint. The drawback to this approach is that it does not provide a reliable comparison when one of the solvers visits a small number of waypoints (or even 0). Therefore, the following two features have been analysed:

- *Efficacy*: number of waypoints visited, on average. The higher this value, the better the solution provided.
- *Efficiency*: ratio $r_i = t_i/w_i$, but only for those matches where w_i equals the number of waypoints of the map. The goal is to minimize this value, as it represents faster drivers.

5.2 Results and analysis

5.2.1 40 milliseconds

Table 1 shows the number of waypoints visited on average, including the standard error. The first notable aspect is

[2]All experiments were performed in a dedicated server Intel Core i5 machine, 2.90GHz 6MB, and 4GB of memory.
[3]www.ptsp-game.net

that, out of the algorithms presented here, when no macro-actions are provided ($L = 1$), only MCTS is able to solve the problem up to a point. The other three algorithms cannot complete the game, not getting more than one waypoint in some rare cases. When the length of macro-action is increased to 5, on average all algorithms achieve close to all waypoints. Excellent efficacy is obtained where the macro-action length is raised to 15, which seems to be the overall best value for this parameter. In general, the GA (especially with high mutation rates) seems to be the algorithm that obtains the best efficacy.

Regarding efficiency, Figure 3 shows the ratio average with standard error obtained with the GA (very similar results are obtained with GAC). One of the first things to note is that the mutation rate that obtains the best solutions is 0.5, for both evolutionary algorithms, especially when the macro-action length is 10 or lower. It is interesting to see how the distinct mutation rates make the algorithm behave differently: it is clear that a higher mutation rate works better with longer macro-actions, obtaining similar results to 0.5. On the other hand, although rate 0.5 is still the best, 0.2 provides better results than 0.8 for smaller values of L. This is confirmed by a Mann-Whitney-Wilcoxon statistical test (MW-test) run on this data: GA-0.5 is better than GA-0.8 for $L = 5$ and 10 (p-values of 0.006 and 0.041, respectively), while GA-0.5 is faster than GA-0.2 for higher values of L (p-values of 0.046 and 0.022 for $L = 15, 20$).

Figure 4 shows the ratio average and standard error of RS, MCTS, GA and GAC (both with mutation rate 0.5). One initial result to note is that the 2012 PTSP winner algorithm (MCTS) is no better than GA or GAC in any of the macro-action lengths tested (with the exception, as mentioned earlier, of $L = 1$, which obtains a ratio of 168.37 ± 7.7). Actually, although overall there is no statistical difference, GA and GAC get better results than MCTS in some of the maps tested (see Section 5.3). It is also worthwhile mentioning that $L = 15$ provides the best results for all algorithms. In fact, these results are compatible with previous studies: Powley et al. [14] used $L = 15$ in their MCTS entry that won the 2012 WCCI competition.

Finally, we observe an interesting phenomenon: while RS is clearly worse than the other algorithms in lower macro action-lengths ($L = 5, 10$), the performance of all algorithms converges to the same performance when the maximum value of L is reached.

A possible explanation for this effect is as follows: for the maximum macro-action length ($L = 20$), the length of the solution (or individual) that is being evaluated is $N = 6$, providing $6^6 \approx 3 \times 10^5$ combinations. This evaluation performs single actions during $L \times N = 120$ time steps, but remember that only the first macro-action is finally performed in the actual game, discarding the rest of the individual actions (as explained in Section 4.1). Clearly, not all the actions in the sequence of 120 moves are equally important; actions chosen at the beginning of this sequence produce a higher impact in the overall trajectory followed by the ship. Therefore, it is fair to assume that, for instance, the first half of the individual (60 actions) is more relevant for the search than the second. In the case of $L = 20$, this is encoded in the first 3 genes, which provides $6^3 = 216$ different possibilities. This is a very reduced search space where even random search can find a good solution (especially considering the amount of evaluations performed per cycle, as shown in Section 5.4).

In the cases where $L = 5$ or $L = 10$, the same amount of look ahead (60 single actions) is obtained with 12 and 6 genes, which provides a search space of $6^{12} \approx 2 \times 10^9$, and $6^6 \approx 3 \times 10^5$ respectively. In these more complex scenarios, the search space is better explored by both GAs (and MCTS) than by RS.

5.2.2 80 milliseconds

All the experiments described above were repeated with the decision time doubled to 80ms. For the sake of space, these results are not detailed here, but they are consistent with the results obtained from spending 40ms per cycle.

All the algorithms perform slightly better when more time is permitted, as would be expected. For instance, GA-0.5 obtains the best performance when $L = 15$, with a ratio of 138.17 ± 1.56, about three units faster than the results achieved in 40ms. Again, the relationship between the different algorithms is maintained: all algorithms perform better with $L = 15$ and their performances get closer as the size of macro-action approaches 20.

5.3 Results by map

These results can also be analysed on a map-by-map basis. Figure 5 represents the ratio achieved by each algorithm in every map of this setup (remember lower is better). First of all, it is interesting to see how the overall performance of the algorithms changes depending on the map presented. For

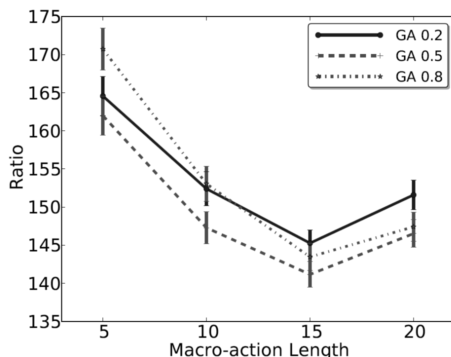

Figure 3: Ratio (time/waypoints) results for GA.

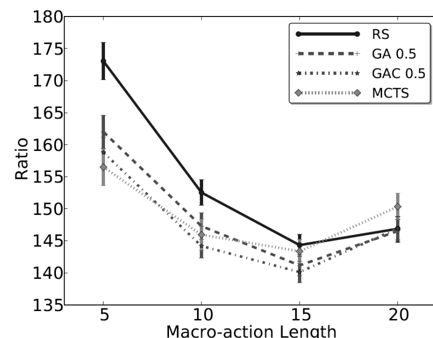

Figure 4: Ratio (time/waypoints) results.

	Average Waypoints Visited							
L	MCTS	RS	GA 0.2	GA 0.5	GA 0.8	GAC 0.2	GAC 0.5	GAC 0.8
1	7.66±0.25	0.03±0.01	0.03±0.01	0.05±0.01	0.01 ± 0.01	0.03 ± 0.01	0.02 ± 0.01	0.01 ± 0.01
5	9.62±0.11	9.76±0.08	9.87±0.06	9.89±0.06	9.86 ± 0.06	9.91 ± 0.05	9.9 ± 0.05	9.69 ± 0.09
10	9.62±0.12	9.95±0.03	9.96±0.03	10.0 ± 0.0	10.0 ± 0.0	9.96 ± 0.04	9.91 ± 0.06	9.96 ± 0.03
15	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0
20	9.97±0.03	9.95±0.04	10.0 ± 0.0	10.0 ± 0.0	10.0 ± 0.0	9.97 ± 0.02	9.98 ± 0.02	9.92 ± 0.04

Table 1: Waypoint visits with 40ms per time step.

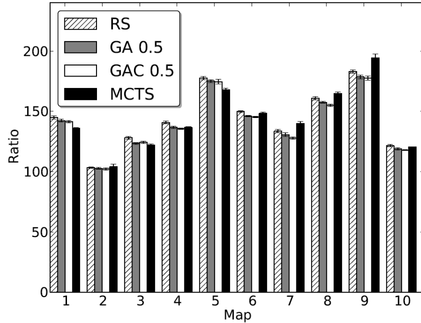

Figure 5: Results per map, $L = 15$.

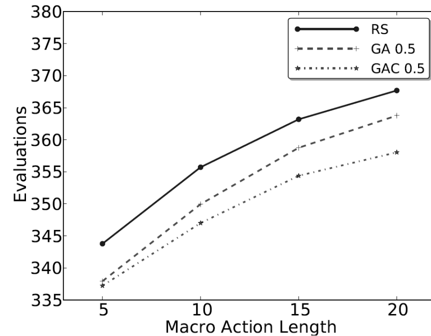

Figure 6: Evaluations per algorithm and L.

instance, map 9 (the one depicted in Figure 1) happens to be especially complex. All algorithms obtain worse results in this map than, for instance, in map 10, that contains no obstacles except the boundaries of the map. Regarding averages, GAC-0.5 obtains the best average of ratio per map in 7 out of the 10 different maps, whilst MCTS leads the rest.

It is also possible to take the best results obtained in each map and compare them with the ones obtained in the WCCI PTSP competition. For instance, both GA and GAC obtain better solutions than the winner of the WCCI competition in 4 out of the 10 maps from the ones distributed with the competition framework. Similarly, MCTS outperforms the best results in two different maps.

The PTSP competition is quite strict regarding controllers that overspend the 40 ms of time allowed per cycle, executing the action 0 (no acceleration, no steering) when this occurs. In the experiments presented in this paper, however, and for the sake of simplicity, this penalisation is ignored. In other words, controllers do use 40ms, but should this time limit be violated (which may happen if a single evaluation takes more time than expected) the desired action is executed anyway. In order to verify if the results obtained are comparable with those from the competition, statistics relating to the number of times the controllers overspent the real-time limit were extracted. This phenomenon occurs in only between 0.1% and 0.03% of the total game cycles in a game, which suggests that, even if the results cannot be fairly considered as new records for those maps, the performance achieved by the algorithms presented in this paper is significant and is comparable to some extent.

5.4 Number of evaluations

Another aspect worth mentioning is the number of evaluations (or calls to the score function) performed every game cycle. MCTS is able to perform more evaluations per cycle, going from about 950 when $L = 5$ to 3000 if $L = 20$, while the other algorithms range between 330 and 370. The main reason for this being that MCTS stores intermediate states of the game in the tree, and is able to reuse this information during the simulations performed in a game cycle, whereas the other three algorithms execute all the actions encoded in each individual, from the first one to the last, producing a more expensive and hence slower evaluation.

The difference between both GAs and RS depends on the overhead of the algorithm (and not on the evaluation itself, which takes the same time for all of them): as shown in Figure 6, RS is the more efficient because of its simplicity, followed by GA and then GAC, which is the slowest due to the cost of using selection and crossover. $MCTS$ is not included in this picture for the sake of clarity.

It is interesting to see how both GAs obtain equivalent (sometimes better) solutions than MCTS, the winner of the 2012 PTSP competition, even when they execute much fewer evaluations per game cycle.

6. CONCLUSIONS

This paper compared the performance of several different search techniques for the PTSP, a real-time game where an action must be supplied every 40ms. The PTSP resembles many modern games where the players need to move providing actions within a few milliseconds, which allows to extrapolate these conclusions to that type of games. The algorithms considered were random search (RS), two different evolutionary algorithms (GA and GAC) and Monte Carlo Tree Search (MCTS). For this, a rolling horizon version for the evolutionary algorithms is introduced. The search space is by using macro-actions, where each one simply repeats a

low-level action a specified number of times, as has previously been shown to be successful [14]. The experiments were performed on the maps distributed with the PTSP competition enabling a direct comparison with the competition's results.

MCTS was the only algorithm able to produce relatively good solutions when no macro-actions were used. The fact that this algorithm stores information in the tree allows for a more efficient search when the search space is vast. However, the best results were obtained using macro-actions, more specifically where a macro-action length of 15 was used. In this case, GAC and MCTS obtained better solutions on each one of the maps, in some cases achieving better results than those from the PTSP competition. The concept of macro actions is in no way restricted to the PTSP and may equally well be applied to the majority of real-time video games. Examples are high level commands in strategy games of first person shooters, where the use of macro-actions instead of low level orders enhances the performance of search algorithms due to a reduction of the search space.

Another important contribution of this paper is understanding how different discretizations of the search space affect the performance of the algorithms in this game. On the one hand, if the granularity is fine (small macro-action lengths, such as $L = 5, 10$), the search algorithm can benefit from artefacts such as the genetic operators. On the other hand, if the granularity is coarse the fitness landscape is rough, and the rolling horizon effect enables random search to perform as well as the other algorithms. Finally, it is worth mentioning that both GAs obtained very good solutions while performing fewer evaluations than the other techniques. This would suggest that the search for solutions is more efficient in these two algorithms. This is significant, suggesting that evolutionary algorithms offer an interesting alternative to MCTS for general video game agent controllers when applied in this on-line rolling horizon mode.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] R. Beer and J. Gallagher. Evolving Dynamical Neural Networks for Adaptive Behavior. *Adaptive behavior*, 1(1):91–122, 1992.

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

[3] P.-A. Coquelin, R. Munos, et al. Bandit Algorithms for Tree Search. In *Uncertainty in Artificial Intelligence*, 2007.

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

[5] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, 2006.

[6] F. Gomez and R. Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1356–1361. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

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

[8] 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 Transactions on Computational Intelligence and AI in Games*, 1(1):73–89, 2009.

[9] S. Matsumoto, N. Hirosue, K. Itonaga, K. Yokoo, and H. Futahashi. Evaluation of Simulation Strategy on Single-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, volume 3, pages 2086–2091, Hong Kong, 2010.

[10] J. Mercieca and S. Fabri. Particle Swarm Optimization for Nonlinear Model Predictive Control. In *ADVCOMP 2011, The Fifth International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 88–93, 2011.

[11] D. Perez, P. Rohlfshagen, and S. Lucas. Monte-Carlo Tree Search for the Physical Travelling Salesman Problem. In *Proceedings of EvoApplications*, 2012.

[12] 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.

[13] 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.

[14] E. Powley, D. Whitehouse, and P. Cowling. Monte Carlo Tree Search with Macro-actions and Heuristic Route Planning for the Physical Travelling Salesman Problem. In *Proc. IEEE Conference of Computational Intelligence in Games*, pages 234–241, 2012.

[15] C. R. Reeves and J. E. Rowe. *Genetic Algorithms-Principles and Perspectives: a Guide to GA Theory*, volume 20. Kluwer Academic Pub, 2002.

[16] D. Robles and S. M. Lucas. A Simple Tree Search Method for Playing Ms. Pac-Man. In *Proceedings of the IEEE Conference of Computational Intelligence in Games*, pages 249–255, Milan, Italy, 2009.

[17] S. Samothrakis and S. Lucas. Planning Using Online Evolutionary Overfitting. In *UK Workshop on Computational Intelligence*, pages 1–6. IEEE, 2010.

[18] S. Samothrakis, D. Robles, and S. M. Lucas. Fast Approximate Max-n Monte-Carlo Tree Search for Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(2):142–154, 2011.