The 2013 Multi-Objective Physical Travelling Salesman Problem Competition

Diego Perez, Edward Powley, Daniel Whitehouse, Spyridon Samothrakis, Simon Lucas, Peter Cowling

***Abstract*— This paper presents the game, framework, rules and results of the Multi-objective Physical Travelling Salesman Problem (MO-PTSP) Competition, that was held in the 2013 IEEE Conference on Computational Intelligence in Games (CIG). The MO-PTSP is a real-time game that can be seen as a modification of the Travelling Salesman Problem, where the player controls a ship that must visit a series of waypoints in a maze while minimizing three opposing goals: time spent, fuel consumed and damage taken. The rankings of the competition are computed using multi-objective concepts, a novel approach in the field of game artificial intelligence competitions. The winning entry of the contest is also explained in detail. This controller is based on the Monte Carlo Tree Search algorithm, and employed Covariance Matrix Adaptation Evolution Strategy (CMA-ES) for parameter tuning.**

I. Introduction

Game competitions have regularly appeared in artificial intelligence (AI) conferences within the last decade. Apart from being excellent benchmarks for comparing different algorithms in similar scenarios, they are very useful for educational purposes, as many research or graduate students have the opportunity to initiate themselves in the field of artificial intelligence. Assuming these competitions are managed properly, they provide testbeds where participants can test their algorithms in an independent and centralized system (this is provided by neutral competition organizers), supplying the basis to draw conclusive results.

In the last few years a wide variety of games have been used as the basis for competitions at many conferences, providing a number of distinct challenges for AI. Among many others, some of them have been particularly successful: *Ms. Pacman vs. Ghost* competition [1] allowed participants to take part in the popular game of *PacMan* both as the player and the ghost team, exploring agent coordination in the latter case. The *Starcraft* competition [2] tackled problems like unit management and partially observable game states, and the popular *Car Racing* Competitions [3] have, as one of their most important features, the player dealing with other drivers.

The purpose of the competition presented in this paper is to cover one of the aspects that had not been analyzed before in AI real-time game competitions: multi-objective optimization. The game proposed for the competition requires the optimization of three different objectives, some of them clearly opposing others. The outcome of the research in this field would not only benefit games like the one proposed here, but also other *single-objective* games where multi-objective approaches could also be taken. As an example, real-time strategy games could benefit from such approaches, as agents usually have to deal with simultaneous tasks: attacking/defensive unit production, resource gathering and world exploration. Additionally, this paper introduces a new way of ranking entries in real-time game competitions, by attending to multi-objective concepts such *Pareto fronts* and *Pareto dominance*.

This paper is organized as follows: Section II explains the game used in the competition. Then, Section III details the game framework and how controllers for this game can be created. Later, Section IV defines the competition, its different tracks and rules. The results of the contest are discussed in Section V. Section VI explains the approach taken by the competition winner and, finally, Section VII concludes this paper with a final evaluation of the competition.

II. The Multi-objective Travelling Salesman Problem

The Multi-Objective Physical Travelling Salesman Problem (MO-PTSP) is a modification of the Physical Travelling Salesman Problem (PTSP), previously introduced by Perez et al. [4] for the WCCI 2012 PTSP Competition. The PTSP is a game where the player controls a ship with the goal of visiting 10 waypoints scattered around the maze in as little time as possible. MO-PTSP adds two more goals to the game: the waypoints must be visited while spending as less fuel as possible and reducing the damage suffered by the ship. This section specifies the main components of the MO-PTSP, such as the game physics, objectives and maps.

A. Game Physics and the real-time Component

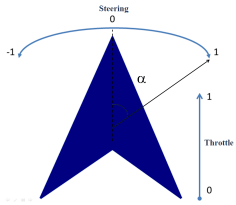
Both PTSP and MO-PTSP are games that share some characteristics. One of these similarities is the physics of the ship used by the player to navigate through the game. In both games, the agent controls a ship where two different inputs can be applied: `steering` and `throttle`. The first input can be set to *left*, *straight* and *right*, while the throttle can be turned *on* or *off*. The combination of these inputs adds up to 6 different actions (see Figure 1) that can be provided at each game step (also referred to here as *tick*, or *game cycle*).

Left and right rotations are performed using a fixed angle, set to $\pi/60$ radians, and the ship acceleration is set to 0.025 pixels per game step squared, values determined by trial an error in order to provide a satisfactory game-play

Diego Perez, Spyridon Samothrakis and Simon Lucas are with the School of Computer Science and Electronic Engineering of the University of Essex (email: {dperez, ssamot, sml}@essex.ac.uk).

Edward Powley, Daniel Whitehouse and Peter Cowling are with the Computer Science department of the University of York (email: {edward.powley, dw830, peter.cowling}@york.ac.uk).

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



Action ID	Steering	Throttle
0 (*Neutral* action)	*Straight* (0)	*Off* (0)
1	*Left* (-1)	*Off* (0)
2	*Right* (1)	*Off* (0)
3	*Straight* (0)	*On* (1)
4	*Left* (-1)	*On* (1)
5	*Right* (1)	*On* (1)

Fig. 1
SHIP INPUTS AND ACTIONS.

experience. The ship keeps its velocity from a game state to the next, making *inertia* a key aspect to take into account when governing the ship. However, the ship would reach a full stop eventually if no acceleration actions are provided, as there is a loss of speed due to *friction*. This friction is set to 0.99 (this is, only 1% of the speed is lost at each step), a value also determined empirically.

The location of the ship in the world can be uniquely determined by three vectors: *position*, that specifies the coordinates of the center of the ship in the maze; *direction*, a vector that indicates where the ship is facing to; and *velocity*, that determines the movement of the ship, including its direction and speed. Note that direction and velocity do not need to be aligned, which in practical terms means the ship could be facing one direction and moving towards another. Given an action, this part of state of the ship is modified as shown in Algorithm 1.

Algorithm 1 Ship update function - no collisions.

function SHIPUPDATE($action$)
  $throttle \leftarrow action$.GETTHROTTLE()
  $steering \leftarrow action$.GETSTEERING()
  $ship.direction$.ROTATES($steering \times steerStep$)
  if $throttle == true$ **then**
    $ship.velocity$.ADD($ship.direction \times shipAcc$)
  $ship.velocity$.MULTIPLY($frictionLoss$)
  $ship.position \leftarrow ship.position + ship.velocity$

The ship can also hit obstacles while it moves within the level, so position and velocity need to be updated differently if the ship's circular bounding collision hits an obstacle in the maze. When this happens, speed is reduced (by a 75% factor) and the velocity vector is modified so the ship bounces off the wall with the appropriate angle. However, the MO-PTSP game introduces a new type of *elastic* obstacle. In the case when the ship hits this kind of obstacle, the reduction of the speed is minor (only 10%), so the player can use this type of walls to change direction of travel abruptly losing less speed than only using throttle and steering. A third type of obstacle (known as *damaging* obstacle), produces a more important decrease in the speed of the ship, reducing it by a factor of 90% (apart from damaging the ship, as explained later).

Another similarity between PTSP and MO-PTSP is the real-time component of the game: the player (also referred here as *agent*, *bot*, or *controller*) must supply an action within a limited budget time, set to 40 milliseconds. This limitation forces the controller to determine the next move quickly, rewarding those agents that are able to plan faster and explore the action search space in a more efficient manner.

While the PTSP was a bit more permissive, MO-PTSP imposes disqualifications in case this time limit is severely violated. In the original game, a neutral action was applied if the controller took more than 40 milliseconds to provide an action, but no other consequences were derived from this misbehaviour. The game was then susceptible to cheating (from a real-time point of view, this stops the continuity of the game, which is not acceptable), as an agent could employ as long as it needs to plan ahead while the only consequence would be a neutral action in a single game tick.

MO-PTSP tackles this situation by imposing a second time limit, set to 120 milliseconds. In case this is violated, it causes the end of the game by disqualifying the player. Although controllers could still potentially spend more than 40 milliseconds (with the same neutral move performed as a penalty), the risk of being disqualified, and the limited gain that could be obtained by the extra milliseconds spent, discourage participants to performed this trick.

B. A Multi-Objective Approach

The main difference with respect to the original game, the PTSP, can be found in the three different objectives that need to be minimized: 1) **Time**: the player must collect all waypoints scattered around the maze in as less time steps (or game cycles) as possible; 2) **Fuel**: the fuel consumed at the end of the game must be minimized; and 3) **Damage**: the ship should end the game with as little damage as possible.

It is very important to stress that all waypoints must be visited in order to consider a game as successfully finished. Otherwise, a very simple but plausible approach for the player can be not to move at all and hence obtain a good result for the other two objectives (no fuel spent and damage suffered). A time-dependant game over condition is established when the ship does not visit a waypoint once a timer has run off. This timer, initially set to 800 time steps, is reduced by 1 at every step if no waypoint is visited, causing the end of the game if it gets to 0.

The ship starts with an initial fuel amount of 5000 units, and one unit is spent every time an acceleration action is performed. Four *fuel canisters* are scattered around the maze, each one providing 250 units of fuel, and visiting a new waypoint also adds 50 fuel units more to the ship. It is important to mention that it is not mandatory to pick these fuel canisters up in order to complete the game, being their collection up to the strategy of the player. In case the amount of fuel available in the ship gets to 0, the agent will not be able to use the throttle any more during that game.

Regarding the third objective, the ship can be damaged by two different game play elements. First, lava lakes are present in the game levels, dealing 1 unit of damage for every game step the ship is driving through them. Secondly, the ship also gets damage when colliding with obstacles (with

the exception of *elastic* obstacles, such normal and *damaging* obstacles. The former type of obstacles (normal walls) inflict 10 units of damage. The latter obstacles, created for this game, produce a more harming effect, dealing 30 damage units. If the player's damage gets up to 5000 units, the ship is destroyed and the game is over.

When the ship is damaged by a collision, it enters in an invulnerable state, where no more collision damage can be dealt to the agent during 50 game steps. This avoids situations when too much damage is suffered by the ship if it is touching an obstacle right after colliding with it.

C. Game Maps

Each level for the MO-PTSP game follows a specific format, employed in several games such as Warcraft, Starcraft or Baldur's Gate, based on the one defined by Nathan Sturtevant. This type of maps have been used by some researchers in the literature before [1]. The levels are stored in ASCII files, where each character determines the type of object or surface located at that pixel in the map grid.

For MO-PTSP, obstacles are represented by the characters 'T', 'D' and 'L', to be used to create normal, damaging and elastic obstacles respectively. Normal surfaces use the character '.', while lava lakes employ ':'. Waypoints and fuel canisters are indicated with 'C' and 'F' respectively, and the player's ship is represented by the character 'S'.

Figure 2 presents an example of a MO-PTSP map, as it is drawn by the framework. Not visited waypoints are depicted as (filled) blue circles, while the already visited ones are shown as empty circles. Green ellipsis are used for the fuel canisters, normal surface is of brown colour, whereas lava is a red-dotted yellow surface. Elastic collisions are blue, normal collisions black and damaging collisions are drawn in red. The ship is drawn as a dark blue polygon, with a green triangle at the back when thrust is being used. The trajectory followed by the ship is drawn with a black line.

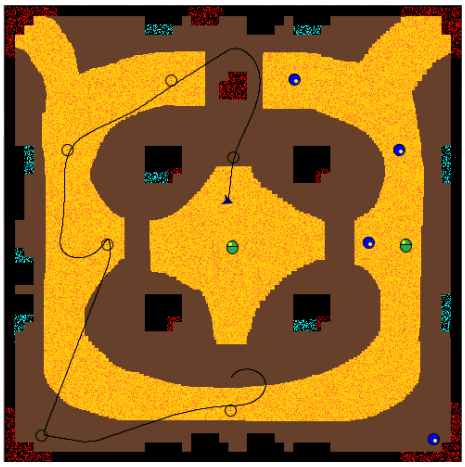


Fig. 2
SAMPLE MO-PTSP MAP.

III. THE MO-PTSP FRAMEWORK

A. Software

The MO-PTSP framework code can be downloaded from the competition webpage [2]. The framework, under the name of *Starter Kit*, contains the code, documentation, and 10 maps to test the controllers in. The software is written in Java, and it is organised in the following packages:

- Package `controllers`: this package contains game bots. The framework also includes several sample controllers to help participants develop their own. Sample controllers are explained in Section III-D.
- Package `framework`: it contains the main code of the game, and it is divided into several sub-packages: `core` package takes care of the logic of game entities, such as waypoints, fuel canisters, ship, etc. `graph`: contains the code for a simple pathfinding tool, that can be used to calculate the shortest path between any pair of points in the map (see Section III-C for more details). Finally, `utils` includes several useful classes for the framework, such as 2D vectors and visual frames.
- Package `wox.serial`: this package contains a sample controller that uses the last version of the Wox Serializer [3]. This is a useful implementation for controllers that use some kind of offline training, such evolutionary algorithms.

Additionally, the package `framework` contains three Java classes that allow the execution of the game:

- `ExecSync.java`: this class allows the execution of the software in three different ways: running a controller in a given map, running it m times in n different maps, and starting the human player mode, where the game is played using the keyboard.
- `ExecFromData.java`: allows the execution of the game in a map initialized through data structures, instead of being read from a file. This mode can be used to execute large number of consecutive runs, useful for reinforcement learning or evolutionary algorithms.
- `ExecReplay.java`: every game played with the framework can have the actions performed by the controller saved into a file. This execution mode allows to replay a game previously saved from one of these files.

B. Game flow

In order to create a controller in this framework, a new class must be created inheriting from the base class `framework.core.Controller`. This class contains one abstract method which contains must be implemented:

```
int getAction(Game, long);
```

This method receives a copy of the current game state and a variable indicating when the controller is due to return an action to execute. When this method is called, this timer is set

[1] `http://movingai.com/benchmarks/dao/`
[2] `http://www.ptsp-game.net/`
[3] `woxserializer.sourceforge.net`

to 40 milliseconds beyond that instant. The value returned by this method should be one of the following 6 action values, from `framework.core.Controller`:

- `ACTION_NO_FRONT`: Throttle off, no steering.
- `ACTION_NO_LEFT`: Throttle off, steer left.
- `ACTION_NO_RIGHT`: Throttle off, steer right.
- `ACTION_THR_FRONT`: Throttle on, no steering.
- `ACTION_THR_LEFT`: Throttle on, steer left.
- `ACTION_THR_RIGHT`: Throttle on, steer right.

Apart from this method, every controller must implement a `public` constructor that receives the same two parameters: a game instance, with the initial state, and a time due. This constructor will be called just before the game starts, and allows the controller to initialize its own data structures and perform some initial planning. The budget time for the constructor of the controller is 1 second, and the game will be over if this time is violated.

Hence, the game flow of MO-PTSP is simple: the framework creates the game, calculates the initialization due time (1 second beyond that instant) and calls the controller's constructor to create the player. If the constructor takes more time than the allowed, it is disqualified from this game. Then, until the game is over, the framework calls the function `getAction` to retrieve the next move to execute, providing the current game state and the time due for the calculation (40 seconds after the `getAction` call). In case this function takes more than 40 milliseconds, the neutral action (`ACTION_NO_FRONT`) is executed. If the call took more than 120 milliseconds, the game is over and the controller is disqualified for the game.

C. Game Interface

Controllers can access information about the current state of the game using several functions available in the `Game` class. Agents can query the game about:

- Waypoints: location of the waypoints (both visited and yet to be collected), number of waypoints visited and how many are left.
- Time: number of time steps since the beginning of the game, number of steps left to visit another waypoint.
- Fuel: remaining fuel in the ship.
- Damage: damage suffered so far in the game.
- Fuel canisters: location of the canisters, both collected and not picked up yet.
- Map locations: for every position in the map, identify the type of obstacle or surface in that location.
- If the game is over or not.

A very important function that `Game` exposes to the controller is `tick(int action)`. This method advances the game state by applying the action indicated as a parameter. By using this function, the agent can foresee the effects in the game of the actions performed, allowing to perform planning in a closed loop manner. It is important to highlight that MO-PTSP is a deterministic game (in other words, one can be certain that applying a move *a* in the game state *s* will always advance to the same next state).

The framework also comes with a simple implementation of a path-finding algorithm that uses A*. This tool can be employed by the agent to obtain paths between every pair of positions in the maze. Only if the participant decides to use it for the controller, an 8-way connected graph is created in the navigable parts of the map with a certain granularity (separation between graph's nodes). The computational time this takes is not detrimental to those participants who prefer not to make use of it, or instead rather to work on a more sophisticated or efficient path-finding implementation.

D. Sample Controllers

The MO-PTSP framework contains several sample bots in order to ease the creation of controllers for the game. In gradually increasing complexity, these samples show how to operate with the different methods and libraries from the framework.

The simplest sample controllers are *RandomController* and *WoxController*, both of them executing actions at random, with the peculiarity that the latter one shows how to operate with the Wox library (as mentioned in Section III-A).

This random behaviour is enhanced in *LineOfSightController*, that navigates towards a certain destination when a waypoint or fuel canister is in line of sight with the ship. The sample *GreedyController* takes care of the waypoints and canisters that are not visible to the ship by means of the pathfinding library included in the framework.

The most complex sample controller is *MacroRSController*. This bot makes use of macro-actions (or repetitions of the same action), a concept exploited in the previous PTSP competition by the winner and other entries. The idea is simple: instead of applying the chosen move in just one game step, it is performed for the next n cycles. This way, the algorithm is able to employ the next n game steps to decide the next sequence of moves to make, increasing the budget time to $40ms \times n$. A stochastic hill climber algorithm is used to search for the actions to apply in the next move.

IV. The MO-PTSP Competition

A. Valid controllers for the competition

The controllers submitted for the competition must run in the framework provided within the *Starter Kit* and, in order to be a valid controller, the following characteristics must be observed: reading from files during execution is allowed, although writing is only possible in the controller's own directory. Multi-threading is not allowed and the process that runs the agent must not use more than 256MB of memory.

The controller must be written in Java. Participants must submit the source code in a zipped file, and the competition server will unzip, compile and run the controller in the appropriate set of maps. The controller must be submitted before the competition deadline, and a short description of the technique employed is expected.

B. Ranking entries

One of the main challenges of a multi-objective competition is to evaluate and rank entries according to its performance. In a single objective scenario, this is straightforward: if the objective of the game is to maximize a score, or minimize completion time (like in the original PTSP), the best result will be the one with the maximum or minimum value that is measured.

However, multi-objective games pose the problem of evaluating objectives in conflict (as they are in this particular case), as happens, for instance, with time and fuel: minimizing completion time would naturally be achieved with a higher speed, which is done by pressing the throttle more often and therefore spending more fuel. In a similar way, very fast controllers would usually follow straight line distances, but this will make the ship drive through lava lakes most of the time, increasing the damage taken.

The competition states that none of these objectives should be preferred to the others. In other words, all these three objectives must be optimized simultaneously. Given a single map, where n results from different competitors have been obtained, the entries are ranked in *Pareto fronts*. A MO-PTSP result can be seen as a triplet $R : (T, F, D)$, where T is the time taken to complete the game, F is the fuel spent and D is the damage taken by the ship at the end of the run. Applying concepts of Multi-Objective optimization, it is said that a result R_1 *dominates* another result R_2 (and it is written $R_1 \preceq R_2$) if:

1) R_1 is no worse than R_2 in all objectives.
2) At least one objective in R_1 is strictly better than its analogous counterpart in R_2.

Dominance establishes a partial order in the results provided. However, there are some cases where it is not possible to say that R_1 dominates R_2 or vice versa (an example in MO-PTSP would be when R_1 is better in the time objective, but R_2 spends a smaller amount of fuel than R_1). When this happens, it is said that these two results belong to the same Pareto front. If there is no result found that dominates a given result R, then R belongs to the *optimal Pareto front*. Thus, it is possible to rank all results in different Pareto fronts, and there will always be an optimal front. There will be no results better than those allocated to the optimal front.

Once the results have been sorted in fronts, points are given to the entries using the following scheme: controllers with no results in the optimal front receive 0 points in this map. 1 point is awarded to each entry that was able to obtain a result that is included in the optimal Pareto front. Finally, the best results for each objective in this map are tracked, and points are awarded to the entries that achieved these records. 2, 9 and 24 additional points are awarded to the entries that obtained respectively 1, 2 and 3 records in a map.

This scheme of points rewards those entries that dominate clearly other contestants. For instance, if a result is better than all others in all objectives, it will be the only member of the optimal Pareto front, receiving a total of 25 $(24+1)$ points in that map, as it obtained the best results in all objectives.

User	Waypoints	Time	Fuel	Damage	Points
PurofMovio	10	**1745**	403	**301**	$9(+1) = 10$
mmorenosi	10	4472	**182**	812	$2(+1) = 3$
epowley	10	2737	227	519	$0(+1) = 1$
macrors	10	1747	761	393	0
Weijia	10	1809	590	485	0

TABLE I
EXAMPLE OF ENTRIES RANKED IN A SINGLE MAP.

A more common scenario is shown in Table I, taken from the final rankings of the competition in one of the starter kit maps. The first three entries receive 1 point as they are in the optimal pareto front, whereas the last two (dominated by at least one of the first three entries) receive no points. The first one obtains the best result in time and damage, so it gets 9 extra points. The second entry gets the best result in fuel, being rewarded with 2 extra points. Finally, the third entry only receives 1 point as it does not achieve any record in this map, but it is still in the optimal front because no other entry dominates it.

Another important aspect to determine is how many runs a controller performs in a given map. On one hand, only a single run (also referred here as a *match*) would penalize potential unlucky cases where the controller behaved poorly, or even did not visit all waypoints. On the other hand, running several matches and calculating the average for each objective would provide misleading results (a bot could try to optimize only one objective per run, and the average would show results that do not match with the real behaviour of the controller in a single match). The competition takes the following approach: each controller is allowed to run a maximum of 5 times on each map, but the evaluation stops as soon as all waypoints are visited in one match. Then, the result of that particular run is used to rank the entry.

Finally, when a controller is evaluated in a set of n maps, the final score is the sum of the points awarded on each one of these levels. To compute the rankings in a set of maps, the entries are sorted according to this total amount of points, and the one with the highest score is declared the winner.

C. Competition Tracks

The MO-PTSP Competition is formed by three tracks.

1) Bot Competition Track: This track evaluates bots submitted to the competition. Before the deadline of the competition, some preliminary rankings are computed in the contest server. Every time a participant submits a controller, it is evaluated on a set of 20 maps, and the rankings of this set are automatically updated in the website. This allows participants to compare between themselves, and get a feeling about how well their controllers would perform for the final rankings.

The set of maps is renewed every few weeks, in order to avoid the controllers to overfit to any particular set. All these maps are taken from a pool of unknown maps, that does not include the levels from the starter kit. The final evaluation, after the deadline of the competition, is run in a new set

of maps, where some are taken from the previous sets and others are completely new to the participants.

2) Human Competition Track: The website allows users to play "as humans" the 10 games from the framework starter kit in an applet. The results are saved and rankings are drawn to determine the best MO-PTSP human player.

3) Human vs. Bot Competition Track: Every time a controller is submitted to the bot competition track, it is also executed in the starter kit maps. Then, a ranking is calculated between the results obtained by bots and humans in these maps, allowing a direct comparison between the two.

V. Results of the competition

The MO-PTSP Competition was held in the 2013 Conference of Computational Intelligence in Games (CIG). The contest received 6 submissions, and the final rankings also included two of the sample controllers explained in Section III-D: *GreedyController* and *MacroRSController*.

Tables II and III summarize the results of the bot and human versus bot competition tracks respectively. As can be seen, the winner of both editions is the same controller, PUROFMOVIO, that is described in detail in Section VI. The full rankings, detailing each one of the matches played, can be seen in the competition website.

In the bot track, the winner achieves 305 points, out of a maximum of 500. The distance with the second ranked bot, WEIJIA, is very significant. Although the first two entries are based on MCTS, the approaches are different: while the winner analyzed the quality of a given game state by a weighted sum of features, the second one incorporated multi-objective measurements in the MCTS algorithm [5]. It is important to notice that the winning entry dominates in all objectives in 7 maps, while is better in two objectives in the other 13 levels.

Regarding the humans versus bot track, PUROFMOVIO was able to beat the winner of the human track (here the second best entry: MMORENOSI). Although the victory is clear, in this case points are more distributed among the first 8 ranked players. A total of 13 human players participated in both the human and the human versus bot tracks.

VI. The winning entry

This section gives a high-level overview of the winning entry, PUROFMOVIO. Further details can be found in [6]. PUROFMOVIO is based on PUROFVIO, the winning entry to the previous (single-objective) PTSP competitions [7], [8].

A. Architecture

The controller has three main components.

1) Distance mapper: During controller initialisation, the shortest-path distance between every waypoint and every other pixel on the map is computed using a scanline flood fill algorithm [9]. This means that during controller execution the distance of a future ship position from the next waypoint can be queried rapidly. The distance mapper takes lava into account when computing distances: moving across a pixel of lava costs the same as moving across $1 + \gamma$ pixels of empty space, where γ is a tunable parameter. An example of a distance map is shown in Figure 3.

Fig. 3
AN EXAMPLE OF A DISTANCE MAP WITH A LAVA WEIGHTING OF $\gamma = 1$. THE CONTOUR LINES CORRESPOND TO DISTANCES A MULTIPLE OF 25 PIXELS FROM THE ORIGIN (SHOWN AS A DOT).

2) Route planner: Also during controller initialisation, the order in which to visit the waypoints is determined. Fuel tanks are also included as "optional" waypoints. The route planner uses classical TSP methods, namely the multiple fragment heuristic [10] and 3-opt local search with first improvement [11], [12]. The edge weights in the TSP graph are the shortest-path distances given by the distance mapper.

3-opt is a hill-climbing algorithm. Starting from an initial path, it considers all triples of edges in the path and all ways in which the path can be reconnected when those edges are deleted. If a resulting path has a lower cost than the initial path, the new path is kept and the process begins again. This continues until no further improvements can be made. In the classical TSP the cost of a path is its length, i.e. the sum of the weights of its constituent edges. In PUROFMOVIO the cost measure is modified to take the physics of MO-PTSP into account: paths which require sharp turns at waypoints or indirect paths between waypoints incur cost penalties.

3) Ship controller: During controller execution, the ship is controlled by Monte Carlo Tree Search (MCTS) [13] with several modifications to deal with the real-time aspect of the problem. Instead of choosing a potentially different action on every time step, the controller instead chooses a *macro-action*, an action to be repeated for the next T time steps. This means that instead of having 40 milliseconds to choose each action, the controller has $40T$ milliseconds to choose each macro-action. This reduces the controller's ability to make fine-grained adjustments to the ship, but is a worthwhile tradeoff.

The second modification deals with the length of the game, which even with macro-actions can result in a decision tree several hundred plies deep. A fixed horizon on

Rank	Bot \ Map	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Total
1	PurofMovio	10	10	25	10	10	10	10	10	10	25	10	10	25	10	25	10	25	25	25	10	**305**
2	Weijia	3	1	0	3	0	3	3	3	3	0	3	3	0	3	0	3	0	0	0	0	**31**
3	MacroRSController	0	3	0	0	3	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	**8**
4	GreedyConroller	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	**3**
5	LordOkami	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
5	age_uc3m	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
5	izzwish15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
5	Yasameer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**

TABLE II

BOT TRACK COMPETITION RESULTS. THE FIRST TWO ENTRIES ARE TWO DIFFERENT VERSION OF MONTE CARLO TREE SEARCH (MCTS) CONTROLLERS. THE TWO SAMPLE CONTROLLERS RANKED THIRD AND FOURTH, AND THE OTHER FOUR PARTICIPANTS, RANKED FIFTH, TOOK THE FOLLOWING APPROACHES, IN THE ORDER SPECIFIED IN THE TABLE: A RULE BASE SYSTEM, A GENETIC ALGORITHM TO TUNE THE SAMPLE GREEDYCONTROLLER, UPPER CONFIDENCE BOUNDS FOR TREES AND, FINALLY ANOTHER MCTS APPROACH.

Rank	Competitor \ Map	Type	1	2	3	4	5	6	7	8	9	10	Total
1	PurofMovio	Bot	3	25	10	10	10	25	3	10	10	1	**107**
2	mmorenosi	Human	0	0	3	3	1	0	3	3	3	3	**19**
3	epowley	Human	0	0	0	0	3	0	0	1	0	3	**7**
4	Weijia	Bot	3	0	0	0	0	0	0	0	0	3	**6**
5	MacroRSController	Bot	0	0	0	0	0	0	3	0	0	0	**3**
6	GreedyController	Bot	3	0	0	0	0	0	0	0	0	0	**3**
7	emsierra	Human	0	0	0	0	0	0	1	0	0	1	**2**
8	LordOkami	Human	0	0	0	0	0	0	0	0	0	1	**1**

TABLE III

HUMANS VERSUS BOT TRACK COMPETITION RESULTS

MCTS is imposed. After d plies, all states are treated as terminal: children are not expanded, simulations are halted. The resulting nonterminal state is evaluated by a heuristic evaluation function. The most crucial terms in the evaluation function are the number of waypoints collected so far, and the distance from the ship to the next waypoint in the route. There are also evaluation terms for fuel and damage, with tunable parameters to control the emphasis placed on these objectives.

The controller may sometimes become trapped in a local optimum, where the search horizon is too short to see a path towards the next waypoint. An example of this is shown in Figure 4. The fitness function gives a reward for approaching the next waypoint but a penalty for damage incurred by driving through lava. The playout length is insufficient for the search to determine that driving through the lava eventually leads to the next waypoint and thus a higher fitness, so the controller instead decides to sit at the edge of the lava until the game times out. A mechanism called *panic mode* is introduced to deal with this: if it is detected that MCTS is failing to find lines of play that make progress towards the goal, the search switches to an alternative evaluation function which more aggressively guides the controller towards the next waypoint (ignoring fuel and damage). The original evaluation takes over again once the controller is unstuck from the local optimum. Panic mode was found to have a limited impact on the time, fuel and damage scores of the controller, but improves its ability to collect all waypoints without timing out.

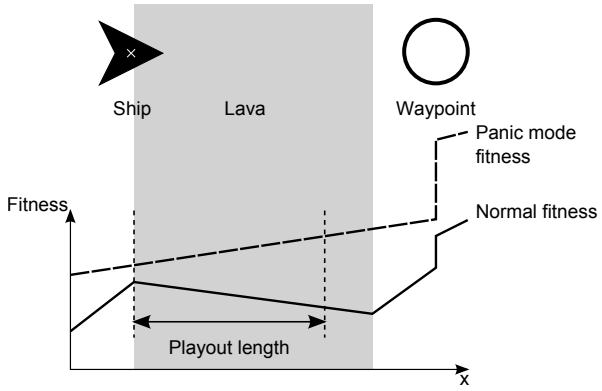


Fig. 4

ILLUSTRATION OF A LOW QUALITY LOCAL OPTIMUM IN A SIMPLIFIED 1-DIMENSIONAL VERSION OF MO-PTSP.

B. Parameter tuning

PUROFMOVIO has 18 tunable parameters in total. Hand-tuning of parameters is challenging: small changes in pa-

rameter values can have large and unpredictable effects on the behaviour of the controller, and the randomised nature of MCTS makes it difficult to assess whether an observed change in behaviour is due to a recent parameter change or simply a lucky or unlucky run.

The parameters were tuned using the *Covariance Matrix Adaptation Evolution Strategy (CMA-ES)* algorithm [14]. CMA-ES optimises a vector of real-valued parameters by repeating three steps: first, a population is sampled at random according to a multivariate normal distribution; second, the population is sorted in order of fitness; third, the fittest individuals are used to skew the distribution used for subsequent samples. As these three steps are repeated, the sampling distribution converges on an optimum.

Since CMA-ES does not use scalar fitness values but only the fitness ranking of the population, it is readily adapted to multiobjective problems [15]. To measure the fitness of a particular parameter vector, the controller is executed once on each of the 10 starter kit maps and the total time, fuel and damage are measured. The population is then ranked by *non-dominated sorting*: individuals on the Pareto front receive rank 1 and are removed; from what remains of the population, individuals on the new Pareto front receive rank 2 and are removed; and so on until the population is exhausted. Within each rank, the individuals are sorted by the sum of time, fuel and damage, with equal weights. Igel et al. [15] suggest more sophisticated tie-breaking schemes, but due to time constraints these were not implemented.

Results of parameter tuning are presented in [6]. Parameter tuning was run three times, obtaining three different sets of parameters. The three tuned controllers outperform our best efforts at hand-tuning, but make slightly different tradeoffs between objectives: for example the third set performs worse on time than the other two, but uses significantly less fuel.

CMA-ES is able to handle noisy and esoteric fitness landscapes, and is explicitly designed not to require tuning. This makes it especially attractive as a "black box" optimisation method;. A mature Java implementation has been made freely available by the algorithm's creators[4], and this implementation was used to tune the parameters for PUROFMOVIO.

VII. CONCLUSION

This paper describes the Multi-Objective Physical Travelling Salesman Problem (MO-PTSP) game and competition, detailing rules, tracks, rankings scheme and results. It also analyzed the winning entry of the competition, a Monte Carlo Tree Search algorithm with an heuristic tuned by CMA-ES.

The multi-objective rankings scheme reflects accurately the quality of the solutions, as the winner's approach performance is much higher than the next best entry. It is also worthwhile mentioning that, in contrast with single-objective rankings, the addition of even a single new result can change profoundly the existing Pareto fronts (e.g. a new result that dominates all others), resulting in a complete reshuffling of competitors.

The competition attracted controllers that employed different techniques, although most of them were variants of MCTS, a technique that has shown its quality and resilience in a wide range of problems both in literature and in different contests (as in the previous WCCI 2012 PTSP Competition). In this particular case, the competition was useful for testing new algorithms that are in development, like the entry by Weijia et al. [5]. Finally, the winning controller showed the benefits of MCTS as a real-time planning and control algorithm in conjunction with a number of add-ons, as the importance of dividing long term and short term planning (waypoint order versus navigation), coarsening the action space using macro-actions, and the use of evolutionary algorithms to tune the parameters of the heuristics employed. Possible promising future research avenues include learning these add-ons automatically and not incorporating them by hand.

[4] https://www.lri.fr/~hansen/

REFERENCES

[1] P. Rohlfshagen and S. M. Lucas, "Ms Pac-Man versus ghost team CEC 2011 competition," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2011, p. to appear.

[2] B. Weber, M. Mateas, and A. Jhala, "Building Human-Level AI for Real-Time Strategy Games," in *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems.*, 2011, pp. 1–8.

[3] D. Loiacono, P. L. Lanzi, J. Togelius, E. Onieva, D. A. Pelta, M. V. Butz, T. D. Lönneker, L. Cardamone, D. Perez, Y. Saez, M. Preuss, and J. Quadflieg, "The 2009 Simulated Car Racing Championship," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2:2, pp. 131–147, 2010.

[4] 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.

[5] W. Wang and M. Sebag, "Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search," *Machine Learning*, vol. 92, pp. 403–429, 2013.

[6] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte Carlo Tree Search with Macro-Actions and Heuristic Route Planning for the Multiobjective Physical Travelling Salesman Problem," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, pp. 73–80.

[7] ——, "Monte Carlo Tree Search with macro-actions and heuristic route planning for the Physical Travelling Salesman Problem," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 234–241.

[8] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. M. Lucas, "Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions," *IEEE Trans. Comp. Intell. AI Games (submitted)*, 2013.

[9] H. Lieberman, "How to color in a coloring book," *ACM SIGGRAPH Comput. Graph.*, vol. 12:3, pp. 111–116, 1978.

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

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

[12] 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.

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

[14] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evol. Comp.*, vol. 9:2, pp. 159–195, 2001.

[15] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evol. Comp.*, vol. 15:1, pp. 1–28, 2007.