The Physical Travelling Salesman Problem: WCCI 2012 Competition

Diego Perez
School of Computer Science
and Electronic Engineering
University of Essex
Colchester CO4 3SQ, UK
Email: dperez@essex.ac.uk

Philipp Rohlfshagen
School of Computer Science
and Electronic Engineering
University of Essex
Colchester CO4 3SQ, UK
Email: prohlf@essex.ac.uk

Simon M. Lucas
School of Computer Science
and Electronic Engineering
University of Essex
Colchester CO4 3SQ, UK
Email: sml@essex.ac.uk

***Abstract*—Numerous competitions have emerged in recent years that allow researchers to evaluate their algorithms on a variety of real-time video games with different degrees of complexity. These competitions, which vary from classical arcade games like Ms Pac-Man to racing simulations (Torcs) and real-time strategy games (StarCraft), are essential to establish a uniform testbed that allows practitioners to refine their algorithms over time. In this paper we propose a new competition to be held for the first time at WCCI 2012: the Physical Travelling Salesman Problem is an open-ended single-player real-time game that removes some of the complexities evident in other video games while preserving some of the most fundamental challenges. This paper motivates and outlines the PTSP and discusses in detail the framework of the competition, including software interfaces, parameter settings, rules and details of submission.**

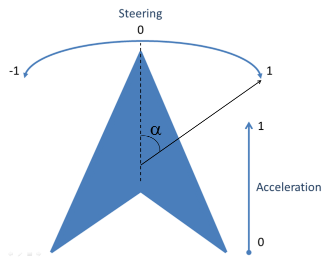
I. An Introduction to the PTSP

Research in game AI has traditionally focussed on 2-player turn-taking board games of perfect information such as Chess and Go, and in many cases has produced algorithms capable of super-human play. Driven by past successes, practitioners have started to concentrate on more complex domains, including those characterised by hidden information, stochasticity, real-time elements and simultaneous moves, to name but a few. Amongst the wide variety of games considered, real-time video games are of particular interest, both academically and commercially: video games are immensely popular yet also notoriously complex and so far, efforts to produce convincing non-player characters (NPCs) have achieved some success but have been labour-intensive; there are some examples where modern Artificial Intelligence (AI) techniques have been applied to video games, as probabilistic techniques in the Halo series, or planning in FEAR and Skyrim. However, most successful video games continue to rely on scripts, animations, finite state machines (often hierarchical) or behaviour trees to convey a sense of intelligence to the gamer. In an ongoing effort to address this issue and offer a machine-learning based alternative, numerous video games competitions have been proposed in recent years, including, for instance, racing simulations (TORCS), Ms Pac-Man, Tron, Planet Wars and the Mario AI Challenge. Irrespective of any potential game applications, they are also interesting problems for academic study.

In this paper we propose a new competition centred around the Physical Travelling Salesman Problem (PTSP), a modification of one of the most well-known combinatorial optimisation problems, the Travelling Salesman Problem (TSP). The goal of the competition is to allow practitioners to compete in a domain that is less complex than most other game-based competitions while retaining some of the essential aspects of modern video games. In particular, the PTSP requires real-time navigation of a single-point mass, an abstraction of a concept found in many modern video games. Despite its simplicity, the PTSP bears significant resemblance to some older video games, including CrystalQuest, XQuest and Crazy Taxi.[1]

The PTSP is a single-player real-time variant of the classical TSP: the TSP consists of a set of spatially distributed cities $n_0, n_1, \dots n_N$. A solution to the TSP is a sequence of cities (*tour*) that visits each city once, returning to the starting city at the end. The *tour length* is the sum of the distances between all consecutive pairs of cities and the goal is to find the tour with the minimum overall length [1]. The TSP may be converted into a single player real-time game: the agent (salesman; herein referred to as a *ship*) and the cities are positioned in two-dimensional continuous space as indicated by their respective coordinates (a pair of real values). The objective of the game is to direct the agent in real-time to visit all the cities (also referred to as *waypoints* in this text) as quickly (i.e., in as few time steps) as possible. This is done by providing an action in each execution step. The actions that govern the ship are summarised in Figure 1. There are two basic commands that can be used: thrusting and left / right rotation. The former can be seen as a boolean input (either the ship thrusts or not) while the latter is an integer value to indicate rotation to the left (-1), to the right (1) or no rotation at all (0). These two inputs may be supplied simultaneously for a total of six different actions that may be carried out at each time step. All actions can be understood as forces applied to the ship that update its position, orientation and speed. The equations shown below

[1]Crazy Taxi and Mario Kart also have a bonus game in which the player drives the car around an open space with the aim of popping a set of balloons, or collecting items, within a given time limit; the PTSP is very similar to this.



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

Fig. 1. Ship forces and actions.

illustrate how these values are updated:

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

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

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

where d_t, v_t and p_t are the vectors that represent the orientation (direction), velocity and position of the ship at time t; α is the rotation angle and L represents the friction factor, used to reduce the speed of the ship at every time step, K is an acceleration constant and the value of T_t depends on the force applied in the step t: 1 if the action involves acceleration, and 0 otherwise. All these values have been tuned manually to create the desired game physics (the specific values used in the competition are shown in section III-F).

The difficulty of the PTSP may be increased by introducing obstacles that the ship must circumvent. Although there is no damage to the ship in the current version of the game, both the ship's velocity and position are affected by collisions. To achieve this, the ship's velocity vector is altered by a collision factor, reducing its speed significantly. An example of a PTSP map that includes obstacles can be seen in Figure 4.

There are two major difficulties inherent to the PTSP. The first relates to the optimal order of waypoints and the second relates to the actual navigation of the ship. While these problems may be addressed separately, it is clear that their interdependency could favour a more integrated approach. It is very important to keep in mind that the physics of the PTSP make this game significantly different to the TSP. In particular, a route connecting all the waypoints whose distances is minimised may not be optimal with respect to the navigation of the ship (if, for instance, the route includes too many changes of direction): while distance alone determines the quality of a solution in the TSP, solution quality in the PTSP corresponds to the time taken to traverse said route and being able to travel at higher speeds generally achieves this.

II. Relevant Research

As seen in Section I, the PTSP can be understood as a combination of two major problems: route optimisation and navigation in real time. This section presents an analysis of the literature related to the TSP, navigation and real time games.

A. TSP

Many different approaches have been proposed in the last few decades to address the TSP and a wide array of heuristics have emerged. An important category of these heuristics are the *tour construction heuristics*, which focus on a local search of available cities. A commonly used heuristic of this kind is the *Nearest Neighbour* heuristic (as described by J. Rosenkrantz et al. in [2]), which just adds to the tour the nearest unvisited city. This should not be confused with the *Greedy* heuristic (which J. L. Bentley presents in [3]), where the tour is built in a similar way, but takes into account that the resultant sequence at each step must be a Hamiltonian cycle where every city has a degree of 2. Another tour construction heuristic is the *Clark-Wright Savings* algorithm, first introduced by G. Clarke and J. Wright [4]. It consists of picking a starting node n_0, and calculating the savings of each pair $(n_i, n_j) : i, j \neq 0$, such that $s(n_i, n_j) = d(n_0, n_i) + d(n_0, n_j) - d(n_i, n_j)$. These savings are then sorted in a descending order until a path is produced in a way that goes exactly once through each node different than n_0. Finally, the path is completed by adding n_0 at each end. A detailed description of this algorithm can be found at [5], by A. M. Frieze.

Insertion Algorithms offer a different type of tour construction. In this case an unvisited city is inserted into the tour between two already included cities, always trying to minimise the added cost caused by this insertion. The tour is built when all cities are in the sequence. The algorithm and its performance may vary depending on the policy chosen to pick the new city to include (see J. Rosenkrantz [2] for details).

Additional heuristics for the TSP include the *k-Opt Heuristics*, which are based on local improvements on existing tours. The most famous ones are 2-Opt (described by G. A. Croes in [6]) and 3-Opt (presented by S. Lin in [7]), where two edges (or three, respectively) are removed from the initial tour and then replaced by different ones that produce a smaller path length.

Meta-heuristics have also been used frequently to tackle the TSP is *Tabu Search* (F. Glover [8] and S. Lin [9] illustrate this technique in their research) using the 2-Opt heuristic: an initial solution to the problem is repeatedly modified by swapping two nodes from the initial route until a stopping criterion has been reached. To ensure the validity of new candidate solutions and to deal with local optima, a list of the edges removed by the algorithm is kept to avoid new paths that produce worse tour lengths. Other techniques used to tackle the TSP include *Simulated Annealing* (by J. S. Kirkpatrick [10]), *Genetic Algorithms*, by R. M. Brady [11] and D. Boese et al. [12], *Ant Colony Optimization* (described by G. L. Dorigo [13]) and using different types of *Neural Networks* (like the ones presented by C. Boeres and V. C. Barbosa [14] or the Kohonen-type neural networks used by F. Favata [15]).

B. Navigation

Navigation has been a highly active field of study in robotics [16]. The problem of navigation (or path planning) typically consists of finding the path, or sequence of positions, that take an agent from an initial location (*start*) to a destination (or *goal*), while avoiding collision with possible obstacles.

Many techniques have been used to approach this problem: *Potential Fields*, described by G. Wang [17] and Y. Hwang [18], are a popular choice because of their simplicity and the good solution quality. This approach builds an artificial potential field, attractive at the goal and repulsive at the obstacles. A mathematical function needs to be defined so that each point in the space is affected by the potential field, in order for the agent to obtain a direction to follow. However, this approach has some limitations, such as the existence of local minima and complications derived from the modelling of an agent as a point-mass.

Other authors prefer to base navigation on the construction of a graph on the map, in a way that its nodes are navigable points in the world. The process of generating a graph is by itself a non-trivial problem ([19], [20]), and it can be tackled from different perspectives, like building visibility graphs, Voronoi diagrams, cell decomposition or navigation meshes (different approaches are illustrated by H. Hale et al. [21] and D. C. Pottinger [22]). Once the graph has been constructed, the agent must use it to navigate through the world. The most common techniques are *Dijkstra* and *A**, described by Russell and Norvig in [23], and provide optimal results. These graph search algorithms often ignore the physics involved in moving an agent, and work on the basis that an agent has no momentum and is free to move to any vacant neighbouring square at each point in the search.

There are other techniques in the literature that navigate the graph using different approaches. A good example is the usage of Rapidly-exploring random trees (or RRTs), by Steven M. LaValle [24]. RRTs are based on a heuristic-driven sampling of the search space, taking the exploration towards unexplored portions of the world. The algorithm is simple and it obtains high quality sub-optimal solutions. It is well suited for high-dimensional search spaces, where it can obtain better results than traditional techniques, if the trade off between quality of the solution and computational effort is taken into account. Monte Carlo Tree Search (MCTS) has been used in the past to solve an obstacle-free version of the PTSP [25]. The authors found that MCTS with a suitable heuristics was able to find solutions to most problem instances, though by visualising the search process it was found that MCTS was performing mostly short-term planning, so the solutions found in this way are unlikely to be close to optimal.

Another important component of navigation is the steering behaviours. In other words, how to specify the commands to effectively move the agent (or robot) through the world. In this case, the benchmark itself defines what types of commands and movements are available. This is defined by the concept of *degrees of freedom*, which is the number of independent parameters that affect the movement, rotation and deformation of the object. Craig W. Reynolds presents in [26] a compilation of steering behaviours for autonomous characters (such as seek, flee, arrive, pursuit or evasion, see Figure 2), giving the agents "the ability to navigate around in a life-like and improvisational manner" [26, p 763].

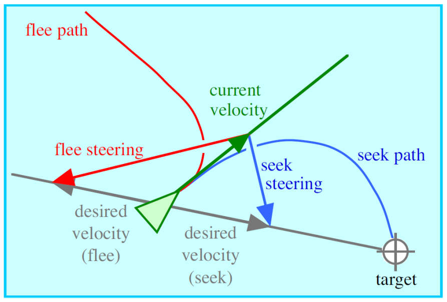


Fig. 2. An example of the steering behaviours *Seek* and *Flee*, from [26].

C. Real-Time Games and Competitions

Many game-related competitions have taken place in the last decade and relevant subset of these competitions involves real-time games. In these competitions the controller must respond with an action within a time limit and must balance the desire to make an optimal move with the necessity to respond quickly. Furthermore, navigation, presented in different ways, is an important part of the controllers programmed for these competitions as can be seen in the following examples.

One of the longest-running real-time competitions is the *Ms Pac Man* screen-capture competition, based on the famous Ms Pac Man arcade game, organised by Simon M. Lucas [27] since 2007. The objective is to control Ms Pac Man and obtain the highest possible score receiving input from a screen capture of the original game. A recent modification of this competition, the *Ms Pac-Man vs Ghosts* competition (by P. Rohlfshagen, D. Robles and S. Lucas [28]), allows the participants to control both Ms Pac-Man and the ghosts. These competitions share a key component with the PTSP: the importance of navigating through the world avoiding obstacles while trying to reach a concrete destination (a pill, an edible ghost or Ms. PacMan when controlling the ghost team).

Another well known competition is the *Simulated Car Racing* competition, by D. Loiacono [29]. It has been running since 2007 and is based on the TORCS racing simulator. With different formats, the goal of this game is to find the fastest driver, in either single- or multi-player modes. Again, the controller has to respond within a given amount of time.

A more recent competition is the *Mario AI* Competition, by S. Karakovskiy and J. Togelius [30]. It is based on the famous video game Super Mario Bros., using a modified version of it called *Infinite Mario Bros.*. Its gameplay track combines the idea of path planning and reactiveness to enemy movements. An example of an entry to this competition, by D. Perez and M. Nicolau [31], used *A** to solve the navigation task and grammatical evolution to evolve behaviour trees to manage the reaction to dynamic enemies. The winner of the 2009 edition, R. Baumgarten [30], also used *A** for his controller, optimising the search in order not to violate the real-time requirements stated in the rules.

Two more recent real-time game competitions are the *2K BotPrize* (by P. Hingston [32]) and the *Starcraft* Competition (by B. Weber [33]). These games cover two genres of games that had not been covered before: First Person Shooter and Real-Time Strategy games, respectively. In both scenarios, navigation through the world plays an important part in the final success of the controllers. In fact, in the case of the 2K BotPrize, the objective is to make the participant's controller behave like a real human (as in the Turing Test), so navigation needs to be not only effective, but also performed in a human-like way.

III. Rules of the Competition

A. Competition Format

Participants of the WCCI 2012 PTSP competition will be able to download the framework from the competition webpage at www.ptsp-game.net where the latest news, important dates and submission instructions may be found also. A *starter kit* is provided in order to ease the creation of controllers. This kit includes the source code for the competition, some sample controllers, and sample maps, that can be used by the participants to train their controllers.

The competition's submission server will be open three months before the submission deadline. From that moment on the server will allow participants to test their controllers against an unknown set of maps that are partially changed over time to avoid overfitting. The web page will present ranking tables for all maps, showing the results of the controllers submitted, providing feedback to the participants as to how their controllers perform, compared with other competitors. Although only one controller is allowed per participant, each participant may resubmit as many times as they like prior to the deadline.

The final evaluation consists of testing all the controllers on a new set of maps. Some of these maps will be picked from the pool of server maps, while others will be completely new. The competition also includes a human track where human players will be able to play the game on different maps and get their scores published in a ranking. This allows a direct comparison between humans and AI-controllers.

B. Evaluation

The objective of the PTSP is to visit all waypoints of a map in the minimum number of time steps and each controller will be evaluated on multiple maps (over multiple trials each) to establish its general performance. A run on a map stops when all waypoints have been visited. However, to avoid indefinite games, the controller has a limit of 1000 time steps to reach the next unvisited waypoint or the game is terminated prematurely. A controller's performance is subsequently established as the average number of waypoints visited and the average time taken to do so: if two or more controllers cover the same number of waypoints, the winner will be the one who required less time.

As the evaluation will take place on multiple maps, the overall winner will be the one that performs best across all of them. To do this, each map will have a ranking of controllers, and points will be awarded depending on their position in the score table. These points will follow the scheme: 10 points for the first classified controller, 8 for the second, 6 for the third, 5 for the fourth and so on up to 1 point for the eighth. The controller with the highest score adding the points of all maps will be the competition winner. In the case of a draw in the final score, the best controller will be considered to be the one with more first positions. If the draw persists, the number of second, third, etc. positions will be taken into account.

C. Submission

Participants must submit the Java source code of their controllers which is then compiled by the organisers. Compilation errors will be reported back to the competitors but execution errors will produce a result of 0 points in the map where the error happened. A brief description of the technique used is also expected. All files must be submitted, in a *zip* file, through the submission webpage before the stated deadline.

D. Software

The Java software framework provided contains the following packages and classes:

- Package controller. All the code for the participant's controller must be self-contained its own package like the sample controllers included: random.RandomController, lineofsight.LineOfSight, greedy.GreedyController and WoxController.WoxController. Section IV gives more details about these sample controllers.
- Package framework. Contains all the game code.
  - Package core. Contains the core code of the game.
  - Package graph. Path-finding code (see section III-G3).
  - Package utils. Includes some useful classes.
  - Several classes for different execution modes as described below.
- Package wox.serial. The framework contains the latest version of WOX Serializer[2], that allows the participant to utilise serialisation of objects into XML files. This is an useful package for those competitors that plan to employ any kind of offline training (as evolutionary or learning algorithms).

The game may be executed in different modes as facilitated by a number of classes created for this purpose:

- ExecSync.java: Synchronous modes, where the game does not monitor the tame taken by the controller to respond. Allows to execute one or several maps, with and without visuals.
- ExecAsync.java: Asynchronous modes, used in the competition to evaluate the controllers.
- ExecReplay.java: To execute replays of old games.
- ExecFromData.java: Special execution modes, as execution in maps created from data or providing a controller through parameters. These modes can be used to execute

[2]woxserializer.sourceforge.net.

large amounts of consecutive runs, which can be useful for reinforcement learning or evolutionary algorithms.

E. Game Flow

To create a controller for this competition, the participants need to create a Java class that extends framework.core.Controller. The class created must include the following: a constructor, that receives a copy of the game state and the time when this controller is due to respond (framework.core.Game, long), and the method getAction(framework.core.Game, long), that receives a copy of the current game state and the time due to respond. This function must return one of the six possible actions, that will be the on executed on that cycle. These actions are defined in the abstract class framework.core.Controller, and are indicated here:

- ACTION_NO_FRONT: No rotation or acceleration.
- ACTION_NO_LEFT: Left rotation but no acceleration.
- ACTION_NO_RIGHT: Right rotation but no acceleration.
- ACTION_THR_FRONT: No rotation, forward acceleration.
- ACTION_THR_LEFT: Left rotation and acceleration.
- ACTION_THR_RIGHT: Right rotation and acceleration.

In the competition execution mode, the controller and the game run in separate threads. This way, the game does not get blocked when the controller takes too much time to respond. The game will apply the action specified by the controller every 40 milliseconds, and it will apply ACTION_NO_FRONT in case the controller runs out of time.

A more complete game flow is shown in figure 3. The Game Thread creates the game and the Controller Thread, and if the initialisation was carried out correctly, the game goes into a main loop that ends when the game is finished (what happens if all waypoints have been visited or the time has run out). During this loop, the game informs the Controller Thread at every cycle about the current game state and the time when the controller is due to respond with an action. The Controller Thread then requests the participant's controller for an action and stores the result in a local variable. Then, after the Game Thread has slept for the action time, it retrieves the last action assigned by the controller and applies it in the game. The Controller Thread is in charge of setting the local variable to the default action every time it is notified by the game that another game cycle starts. This way, should the controller takes more time than allowed processing the next move, this action is retrieved and executed.

It is important to note that the controller has access to the whole game state, so queries can be made as to the condition of the ship and environment. Furthermore, a copy of the Ship object can be accessed from the game state, allowing the performance of simulations by applying actions, before returning the final move.

When the game is ended, the actions executed in this run can be dumped to a file, with a name containing the current date and time. The format of this file is the number of actions, followed by all the executed moves, one per line.

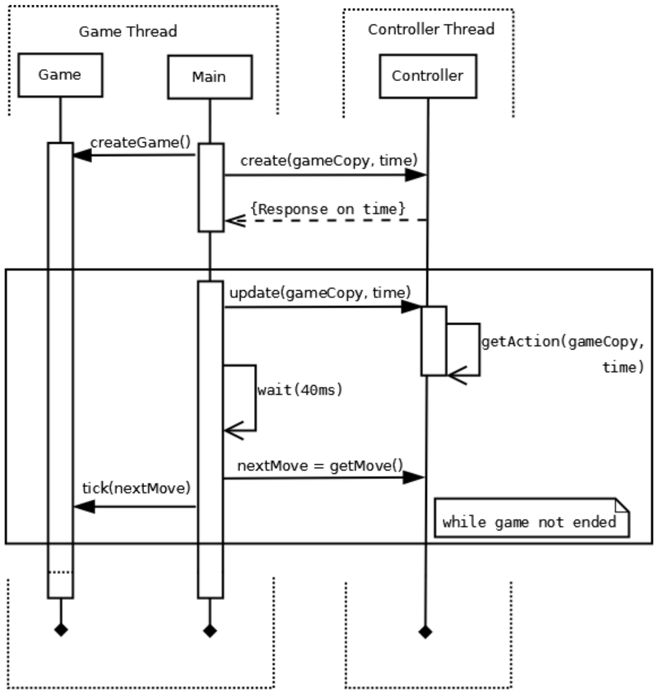


Fig. 3. Game flow.

F. Ship Physics

The game framework implements the physics described in section I. However, the following parameters need to be defined in order to obtain the desired physics for the game:

- Rotation step (α) is set to $\pi/60$ and $-\pi/60$ radians for right and left steering respectively, and 0 for no rotation.
- The friction factor (L) is set to 0.99. It is used to decelerate the ship when no thrust is applied.
- The constant acceleration value (K) is fixed to 0.025.
- The collision factor used to reduce the speed of the agent when it collides with an obstacle is set to 0.05.

G. Controllers

1) Programming specifications: Regarding the correctness of the controllers, the following technical specifications must be observed:

- The controllers must be programmed in Java. Extension to other languages could be considered for future editions of this competition.
- Multi-threading is not allowed.
- Reading from files is allowed at any moment. Writing to files is only allowed if the controller does it in its own directory. At the moment of the final evaluation, all controllers will be executed using the exact same files included by the participants in the packages submitted.
- There are two time limits that must be respected, at the controller initialisation and update. The first one, set to $1000ms$, is the maximum amount of time allowed for the constructor of the controller. The second, set to $40ms$, is the time allowed for the controller to provide an action

to perform at every execution step. If the first time limit is not respected, the controller will not be executed on that map. Should the controller spend more time than allowed during the update call, the action performed by the controller in this step will be action 0 (i.e. no thrust and no rotation).[3]

- The process of the controller is also memory limited, no higher than 256MB. Not respecting this limitation will produce a result of 0 points in that map.
- Copies of the game state are provided to the controllers in order to have full access to the game during execution. Any attempt to modify the real state of the game, other than the required actions to control the ship, will end with the disqualification of the controller from the whole competition. The same applies for any attempt of using a bug in the system to obtain any kind of benefit over other participants.

2) Available data: Table I shows the most important methods and fields available to the controller and the information that they retrieve.

3) Path-finding: The *Starter Kit* provides code for path-finding, that can be used for any controller. This code creates a grid graph on the navigable parts of the map, using eight-way connectivity between the nodes (i.e, each node is connected to its eight neighbour nodes, if present.) The A* algorithm is used to create paths between nodes, and these paths are stored in memory so the computational time is not heavily affected. The sample controller *GreedyController*, also described in section IV-C, shows an example of usage of these classes. It is important to mention that the execution of the code devoted to this end must be triggered by the controller, so that, should the participant decide not to use it, they will not be penalised by the time/storage consumption that this process involves.

H. Maps

The maps accepted by this game are ASCII files, that have the following format: the first two lines must specify the height and the width of the map. Then, the keyword *map* indicates the start of the map. Finally, the following lines compose the map itself. Each position in the map is specified by an ASCII character, which can have different values for walls or obstacles ('@' and 'T'), waypoints ('C'), starting point of the ship ('S') and empty spaces ('.')[4].

An example of one of the maps provided with the benchmark can be seen in Figure 4.

The waypoints are painted with different colours depending on the ship have visited or not (red for non visited, blue for visited, in the actual game). The ship is depicted as a blue polygon, with a green triangle at the back that denotes that the thrust is being pressed. The trajectory followed by the

[3]The competition will be run in a dedicated server Intel Core i5 machine, 2.90GHz 6MB, and 4GB of memory.

[4]This format is based on the symbols used by Nathan Sturtevant, from games such as Warcraft, Starcraft or Baldur's gate. They have been used by many researchers in the literature - http://movingai.com/benchmarks/dao/

Function	**Information**
From Game	
getWaypoints()	Returns the list with all the waypoints in the map.
getWaypointsLeft()	Indicates the number of waypoints that are still to be visited.
getTotalTime()	Gets the time spent since the beginning of the game.
getStepsLeft()	Returns the steps left until the time runs out.
getMap()	Returns the map of the game (instance of Map).
getMapSize()	Returns the dimensions of the map (instance of java.awt.Dimension).
getShip()	Gets the ship of the game (instance of class Ship).
advanceMap()	Advances the current map to the next loaded one.
getCopy()	Gets a copy of the whole game state in a Game object.
From Waypoint	
isCollected()	Indicates if this waypoint has been collected/visited.
Vector2d s	Indicates the position of this waypoint.
RADIUS	Represents the radius (number of pixels) of the waypoints.
From Map	
getMapChar()	Gets a bi-dimensional array with the contents of the current map. Each position is a pixel on the map, and a character of the map file (see section III-H).
getMapHeight()	Gets the height of the map (in pixels).
getMapWidth()	Returns the width of the map (in pixels).
getStartingPoint()	Gets the starting point of the ship.
isObstacle(x,y)	Returns true if there is an obstacle in the position given.
checkObsFree(rect)	Checks if there are no obstacles in a given rectangle.
LineOfSight(o, d)	Checks if there are no obstacles from the origin position to the destination (considering ship radius).
distToCollision(v,w,d)	Returns the distance to a potential obstacle from a given point (v), in a specified direction (w) an up to a maximum distance (d). Gets -1 if no collision.
getCopy()	Gets a copy of the Map object.
From Ship	
getCollLastStep()	Indicates if there was a collision in the last step.
update(action)	Performs the action provided.
getCopy()	Gets a copy of the Ship object.
Vector2d s	Position of the ship.
Vector2d sp	Position of the ship in the previous step.
Vector2d v	Velocity of the ship.
Vector2d d	Direction of the ship (where the ship is facing, not necessarily the same as Velocity).
SHIP_RADIUS	Represents the radius (number of pixels) of the ship.
From Controller	
getThrust(action)	Returns true if the action given accelerates the ship.
getTurning(action)	Returns -1, 1 or 0 if the action given rotates left, right or none, respectively.
getActionFromInput(thrust, turn)	Given an acceleration boolean and a turn sense, returns the desired action identifier.

TABLE I
CODE INTERFACE.

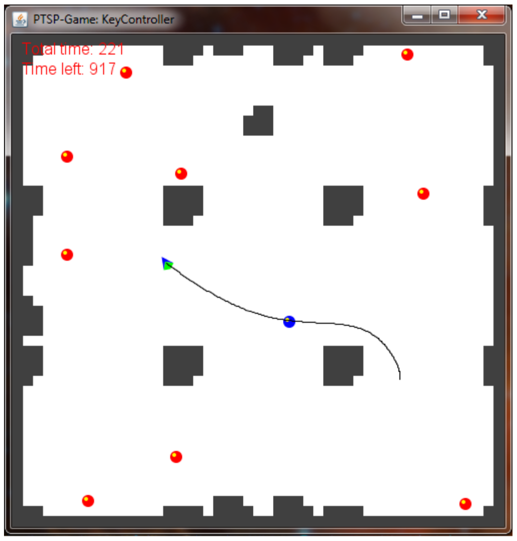


Fig. 4. Sample map, during execution.

ship is shown with a black line, and obstacles are drawn in a dark colour on a white background.

IV. Sample Controllers

A. *RandomController*

This is probably the most basic controller that can be made in this benchmark. This controller returns a random action at every execution step. The class, controllers .random. RandomController, inherits from framework.Controller and it only needs to create a constructor and override the getAction() method from the base, both of them receiving an instance of framework.core.Game. Its code is shown below:

```
Random m_rnd;

public RandomController(Game gameCopy, long due)
{
    m_rnd = new Random();
}

public int getAction(Game gameCopy, long due)
{
    return m_rnd.nextInt(Controller.NUM_ACTIONS);
}
```

B. *LineOfSightController*

This is an extension of the previous controller. The ship in this case wanders randomly through the level but, when the closest waypoint is in line of sight, the ship drives directly to it. This functionality is achieved using the function LineOfSight(origin , destination), from class Map, that checks whether there is an obstacle directly between the origin and destination points.

C. *GreedyController*

This controller is another extension of the LineOfSight controller. In this case, when there is no line of sight to a waypoint, the controller tries to find the action that takes the ship closer to the closest waypoint, avoiding the obstacles in between. It does this by using the path-finding code, as mentioned in section III-G3, calculating the distances from the ship to the waypoints using the cost of the shortest paths. In order to use the path-finding code included in the benchmark, the first step is the creation of the graph. This can be made with a single line:

```
Graph graph = new Graph(gameCopy.getMap());
```

Another useful function of the framework.graph.Graph class is getClosestNodeTo(), that returns the closest node in the map to a given position. This function can be used, for instance, to obtain the closest node to the ship and the waypoints, and hence is able to calculate paths between their positions. The function that calculates paths is getPath(), and returns an instance of the class framework.graph.Path. This is an example of how to create and obtain paths between two nodes:

```
Ship ship = a_gameCopy.getShip();
Waypoint way = a_gameCopy.getWaypoints().get(0);
Vector2d p = ship.s; //Ship position
Vector2d d = way.s; //Waypoint position

Node sNode = graph.getClosestNodeTo(p.x, p.y);
Node wNode = graph.getClosestNodeTo(d.x, d.y);

Path path = graph.getPath(sNode.id(), wNode.id());

double pathCost = path.m_cost;
Vector<Integer> pathNodes = path.m_points;
```

D. *WoxController*

This is a modification of the RandomController. The purpose of this controller is to demonstrate how to use the Wox package included with the distribution (as stated in section III-D). This controller reads a serialized object from a file, that contains certain variables that bias the choose of random actions executed during the game.

E. *Performance of sample controllers*

Table II shows the performance of the first three sample controllers included with the distribution (differences between RandomController and WoxController and insignificant). These results can be seen as a baseline for the competitors. The table also distinguishes between the 10 starter kit maps (the ones distributed with the framework) and the first set of 20 maps from the submission server (as stated in section III-A).

V. Conclusion

This paper proposes a new real-time gaming competition based on the Physical Travelling Salesman Problem, a single-

Maps	**Starter kit (Avg.)**		**Server (Avg.)**	
Controller	**Waypoints**	**Time**	**Waypoints**	**Time**
RandomController	0.14	1071.6	0.15	1054.77
LineOfSightController	3.44	2321.52	3.09	2147.52
GreedyController	9.1	4469.1	7.35	3913.45

TABLE II
Performance of sample controllers. Each controller was executed five times on each map.

player game variant of the well-known combinatorial optimisation problem, the Travelling Salesman Problem. Competitors are required to navigate a ship in real-time to visit a set of waypoints as quickly as possible. The waypoints are scattered randomly across a playing field with numerous obstacles.

The PTSP provides a relatively simple yet interesting and challenging competition on which to test a wide range of optimisation and planning methods. Among these, we are particularly interested to see how Monte Carlo Tree Search, Evolutionary Algorithms, and meta-heuristics such as Ant Colony Optimisation compare with each other. Also, an open question is whether it is best to plan the city order then the detailed action sequence to navigate that route, or whether it is best to simply optimise the action sequence at a single level.

The PTSP is able to stand alone as an interesting optimisation problem, a natural extension of the TSP. Additionally, its game-like setting suggests a number of applications within real-time video games, including the navigation of opponent agents and the automated testing of map difficulty levels.

Finally, the PTSP game is a benchmark that permits many additions and modifications for future versions. For instance, the inclusion of different type of obstacles (elastic, inelastic, damaging, etc.) would change the way the levels are played. Another option would be to add a second player to compete with, making the game much more challenging. An interesting modification would also be to deny the controller access to the level's map, limiting the ship's visibility to enforce a more reactive navigation.

Acknowledgements

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

References

[1] D. S. Johnson and L. A. McGeoch, *The Traveling Salesman Problem: A Case Study in Local Optimization*, E. H. L. Aarts and J. K. Lenstra, Eds. John Wiley and Sons, Ltd., 1997.

[2] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *Society for Industrial and Applied Mathematics Journal on Computing*, vol. 6, pp. 563–581, 1977.

[3] J. L. Bentley, "Fast algorithms for geometric traveling salesman problems," *ORSA Journal on Computing*, vol. 4, pp. 387–411, 1992.

[4] G. Clarke and J. Wright, "Scheduling of vehicles from a central depot to a number of delivery points," *Operations Research*, vol. 12, no. 4, pp. 568–581, 1964.

[5] A. M. Frieze, "Worst-case analysis of algorithms for travelling salesman problems," *Methods of Operations Research*, vol. 32, pp. 97–112, 1979.

[6] G. A. Croes, "A method for solving traveling salesman problems," *Operations Research*, vol. 6, pp. 791–812, 1958.

[7] S. Lin, "Computer solutions of the traveling salesman problem," *Bell Systems Technical Journal*, vol. 44, pp. 2245–2269, 1965.

[8] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers and Operations Research*, vol. 13, pp. 533–549, 1986.

[9] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling salesman problem," *Operations Research*, vol. 21, pp. 498–516, 1973.

[10] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1973.

[11] R. M. Brady, "Optimization strategies gleaned from biological evolution," *Society for Industrial and Applied Mathematics Review*, vol. 26, pp. 804–806, 1985.

[12] K. D. Boese, A. B. Kahng, and S. Muddu, "A new adaptive multi-start technique for combinatorial global optimizations," *Society for Industrial and Applied Mathematics Review*, vol. 26, pp. 101–113, 1994.

[13] M. Dorigo and L. Gambardella, "Ant colonies for the travelling salesman problem," *Biosystems*, vol. 43, no. 2, pp. 73–81, 1997.

[14] M. C. S. Boeres, L. A. V. de Carvalho, and V. C. Barbosa, "A faster elastic-net algorithm for the traveling salesman problem," in *Proceedings of the 1992 IEEE International Joint Conference on Neural Networks*, 1992, pp. 215–220.

[15] F. Favata and R. Walker, "A study of the application of kohonen-type neural networks to the travelling salesman problem," *Biological Cybernetics*, vol. 64, pp. 463–468, 1991.

[16] H. Choset, "Coverage for robotics - a survey of recent results," *Annals of Mathematics and Artificial Intelligence*, vol. 31, pp. 113–126, May 2001.

[17] Y. Wang and G. Chirikjian, "A new potential field method for robot path planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2000, pp. 977–982.

[18] Y. K. Hwang and N. Ahuja, "A potential field approach to path planning," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 1, pp. 23–32, 1992.

[19] G. Dudek, M. Jenkin, E. Milios, and D. Wilkes, "Robotic exploration as graph construction," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 6, pp. 859–865, 1991.

[20] F. Lamarche and S. Donikian, "Crowd of virtual humans: a new approach for real time navigation in complex and structured environments," *Computer Graphics Forum*, vol. 23, no. 3, pp. 509–518, 2004.

[21] D. H. Hale, G. M. Youngblood, and P. Dixit, "Automatically-generated convex region decomposition for real-time spatial agent navigation in virtual worlds," in *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[22] D. C. Pottinger, "Terrain analysis in real time strategy games," in *Proceedings of Computer Game Developers Conference*, 2000.

[23] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[24] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Computer Science Department, Iowa State University, Tech. Rep., 1998.

[25] D. Perez, P. Rohlfshagen, and S. M. Lucas, "Monte-Carlo Tree Search for the Physical Travelling Salesman Problem," in *Proceedings of EvoApps, Malaga, Spain*. Springer, 2012, p. To Appear.

[26] C. W. Reynolds, "Steering behaviors for autonomous characters," in *Proceedings of Game Developers Conference*, 1999, pp. 763–782.

[27] S. M. Lucas, "Ms Pac-Man competition," *SIGEVOlution*, vol. 2, pp. 37–38, December 2007.

[28] 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.

[29] D. Loiacono, P. Lanzi, J. Togelius, E. Onieva, D. Pelta, M. Butz, T. Lönneker, L. Cardamone, D. Perez, Y. Sáez, M. Preuss, and J. Quadflieg, "The 2009 simulated car racing championship," *IEEE Transactions on Computational Intelligence and Games*, vol. 3, no. 2, pp. 131–147, 2010.

[30] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario ai competition," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[31] D. Perez, M. Nicolau, M. O'Neill, and A. Brabazon, "Reactiveness and navigation in computer games: Different needs, different approaches," in *Proceedings of IEEE Conference on Conputational Intelligence and Games*, 2011, pp. 273–280.

[32] P. Hingston, "A new design for a turing test for bots," in *Proceedings of IEEE Symposium on Conputational Intelligence and Games*, 2010, pp. 345–350.

[33] B. G. 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.