Multi-Objective Monte Carlo Tree Search for Real-Time Games

Diego Perez, *Student Member, IEEE*, Sanaz Mostaghim, *Member, IEEE*, Spyridon Samothrakis, *Student Member, IEEE*, Simon M. Lucas, *Senior Member, IEEE*

***Abstract*—Multi-objective optimization has been traditionally a matter of study in domains like engineering or finance, with little impact on games research. However, action-decision based on multi-objective evaluation may be beneficial in order to obtain a high quality level of play. This paper presents a Multi-objective Monte Carlo Tree Search algorithm for planning and control in real-time game domains, those where the time budget to decide the next move to make is close to 40ms. A comparison is made between the proposed algorithm, a single-objective version of Monte Carlo Tree Search and a rolling horizon implementation of Non-dominated Sorting Evolutionary Algorithm II (NSGA-II). Two different benchmarks are employed, Deep Sea Treasure and the Multi-Objective Physical Travelling Salesman Problem. Using the same heuristics on each game, the analysis is focused on how well the algorithms explore the search space. Results show that the algorithm proposed outperforms NSGA-II. Additionally, it is also shown that the algorithm is able to converge to different optimal solutions or the optimal Pareto front (if achieved during search).**

I. Introduction

Multi-objective optimization has been a field of study in manufacturing, engineering [1] and finance [2], while having little impact on games research. This paper shows that multi-objective optimization has much to offer in developing game strategies that allow for a fine-grained control of alternative policies. The application of such approaches to this field can provide interesting results, especially in games that are long or complex enough that long-term planning is not trivial, and achieving a good level of play requires balancing strategies.

At their most basic, many competitive games can be viewed as simply two or more opponents having the single objective of winning. Achieving victory is usually complex in interesting games, and successful approaches normally assign some form of value to a state, value being long term expected reward (i.e. expectation of victory). Due to a massive search space, heuristics are often used for this value function. An example is a chess heuristic that assigns different weights to each piece according to an estimated value.

Multi-objective approaches can be applied to scenarios where several factors must be taken into account to achieve victory. The algorithms can balance between different objectives, in order to provide a wide range of strategies well suited to the different stages of the game being played, or to face existing opponents. Application of such approaches could be real-time strategy games, where long term planning must be carried out in order to balance aspects such as attack units, defensive structures and resource gathering. This should allow smart balancing of heuristics.

This paper proposes a multi-objective real-time algorithm version of Monte Carlo Tree Search (MCTS), a popular reinforcement learning approach that emerged in the last decade. The proposed algorithm is tested in two different games: a real-time version of the Deep Sea Treasure (DST; a classical multi-objective problem), and the Multi-Objective Physical Travelling Salesman Problem (MO-PTSP). The algorithm is also compared with another two approaches: a single-objective MCTS (that uses a weighted sum of features to value the state) and a rolling horizon Non-dominated Sorting Evolutionary Algorithm II (NSGA-II). This paper extends and formalizes our previous work described in [3].

Two main goals can be identified in this paper: first, the proposed algorithm must be applicable to real-time domains (those where the next move to make must be decided within a small time budget) and it should obtain better or at least the same performance than the other state-of-the-art algorithms. It is important to highlight that all three algorithms tested employ the same heuristic functions to evaluate the features of a given game state. Thus, the focus of this research is set on how the algorithm explores the search space, instead of providing the best possible solution to each given problem.

Secondly, the algorithm must be able to provide solutions across the multi-objective spectrum: by parametrizing the algorithm, it must be possible to prioritize one objective over the others and therefore converge to solutions according to these preferences. Therefore, the main contribution of this paper is the proposal of a multi-objective version of MCTS that is applicable to real-time domains and is able to provide different solutions across the multi-objective spectrum.

The paper is structured as follows. First, Sections II and III provide the necessary background for MCTS and multi-objective optimization, respectively. The algorithm proposed in this research is described in detail in Section IV. Then, Section V defines the games used to test the algorithms, with the results discussed in Section VI. Finally, some conclusions and possible extensions of this work are drawn in Section VII.

II. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search algorithm that was originally applied to board games, gaining momentum and popularity in the game of Go. This game is played in a square grid board, with a size of 19×19 in the original game, and 9×9 in its reduced version. The game is played in turns, and the objective is to surround the opponent's stones

Diego Perez, Spyridon Samothrakis, Simon M. Lucas (School of Computer Science and Electronic Engineering, University of Essex, Colchester CO4 3SQ, UK; email: {dperez,ssamot,sml}@essex.ac.uk);
Sanaz Mostaghim (Department of Knowledge and Language Engineering, Otto-von-Guericke-Universitt Magdeburg, Magdeburg, Germany; email: sanaz.mostaghim@ovgu.de)

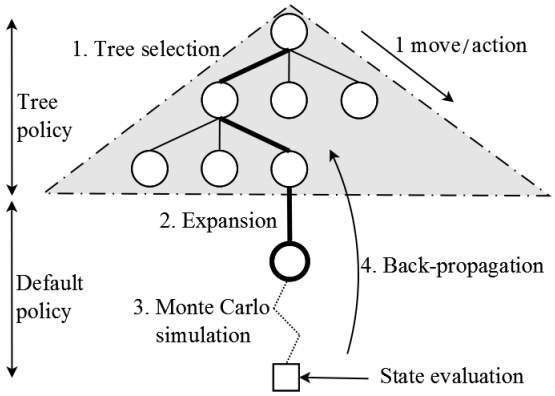


Fig. 1: MCTS algorithm steps.

by placing stones in any available position on the board. Due to the very large branching factor of the game and the absence of clear heuristics to tackle it, Go became object of study for many researchers. MCTS was the first algorithm able to reach professional level play in the reduced board size version [4]. After its success in Go, MCTS has been used extensively by many researchers in this and different domains. An extensive survey of MCTS methods, variations and applications, has been written by Browne et al. [5].

MCTS is considered to be an *anytime* algorithm, as it is able to provide a valid next move to choose at any moment in time. This is true independently from how many iterations the algorithm is able to make (although, in general, more iterations usually produce better results). This differs from other algorithms (such as A* in single player games, and standard Min-Max for two player games) that normally provide the next play only after they have finished. This makes MCTS a suitable candidate for real-time domains, where the decision time budget is limited, affecting the number of iterations that can be performed.

MCTS is an algorithm that builds a tree in memory. Each node in the tree maintains statistics that indicate how often a move is played from a given state ($N(s,a)$), how many times each move is played from there ($N(s)$) and the average reward ($Q(s,a)$) obtained after applying move a in state s. The tree is built iteratively by simulating actions in the game, making move choices based on statistics stored in the nodes.

Each iteration of MCTS can be divided into several steps, as introduced by G. Chaslot et al. [6]: *Tree selection*, *Expansion*, *Monte Carlo simulation* and *Back-propagation* (all summarized in Figure 1). When the algorithm starts, the tree is formed only by the root node, which holds the current state of the game. During the *selection* step, the tree is navigated from the root until a maximum depth or the end of the game has been reached.

In every one of these action decisions, MCTS balances between exploitation and exploration. In other words, it chooses between taking an action that leads to states with the best outcome found so far, and performing a move to go to less explored game states, respectively. In order to achieve this, Kocsis and Szepesvári [7] applied Upper Confidence Bound (UCB1, see Equation 1) as a *Tree Policy*.

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

The balance between exploration and exploitation can be tempered by modifying C. Higher values of C give added weight to the second term of the UCB1 Equation 1, giving preference to those actions that have been explored less, at the expense of taking actions with the highest average reward $Q(s,a)$. A commonly used value for single-player games is $\sqrt{2}$, as it balances both facets of the search when the rewards are normalized between 0 and 1. The value of C is application dependant, and it may vary from game to game. It is worth noting that MCTS, when combined with UCB1, reaches asymptotically logarithmic regret on each node of the tree [8].

If, during the *tree selection* phase, a node has fewer children than the available number of actions from a given position, a new node is added as a child of the current one (*expansion* phase) and the *simulation* step starts. At this point, MCTS executes a Monte Carlo simulation (or roll-out; *default policy*) from the expanded node. This is performed by choosing random (either uniformly random, or biased) actions until the game end or a pre-defined depth is reached, 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 visited, using the reward obtained in the evaluation of the state. These steps are executed in a loop until a termination criteria is met (such as number of iterations).

MCTS has been employed extensively in real-time games. An example of this is the popular real-time game *Ms. PacMan*. The objective of this game is to control Ms. PacMan to clear the maze by eating all pills, without being captured by the ghosts. An important feature of this game is that it is *open-ended*, as an end game situation is, most of the time, far ahead in the future and can not be devised by the algorithm during its iterations. The consequence of this is that MCTS, in its vanilla form, it is not able to know if a given ply will lead to a win or a loss in the end game state. Robles et al. [9] solved this problem by including hand-coded heuristics that guided search towards more promising portions of the search space. This approach enabled the addition of heuristics knowledge to MCTS, as in [10], [11].

MCTS has also been applied to single-player games, like SameGame [12], where the player's goal is to destroy contiguous tiles of the same colour, distributed in a rectangular grid. Another use of MCTS is in the popular puzzle Morpion Solitaire [13], a connection game where the goal is to link nodes of a graph with straight lines that must contain at least five vertices. Finally, the PTSP has also been addressed by MCTS, both in the single-objective [14], [15] and the multi-objective versions [16]. These papers describe the entries that won both editions of the PTSP Competition.

It is worthwhile mentioning that in most cases found in the literature, MCTS techniques have been used with some kind of heuristic that guides the Monte Carlo simulations or the tree selection policy. In the algorithm proposed in this paper,

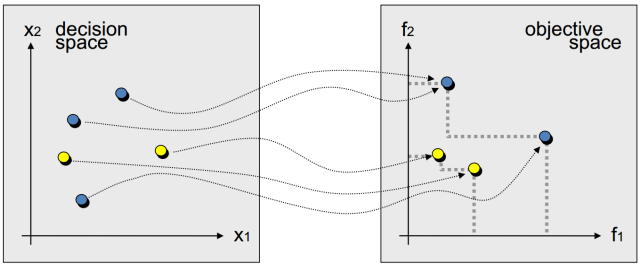


Fig. 2: Decision and objective spaces in a MOP with two variables (x_1 and x_2) and two objectives (f_1 and f_2). In the objective space, yellow dots are non-optimal objective vectors, while blue dots form a non-dominated set. From [**?**].

simulations are purely random, as the objective is to compare the search abilities of the different algorithms. The intention is therefore to keep the heuristics to a minimum, and the existing pieces of domain knowledge are shared by all the algorithms presented (as in the case of the score function for MO-PTSP, described later).

III. Multi-Objective Optimization

A multi-objective optimization problem (MOP) represents a scenario where two or more conflicting objective functions are to be optimized at the same time and is defined as:

$$optimize \quad \{f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x})\} \tag{2}$$

subject to $\vec{x} \in \Omega$, involving $m(\geq 2)$ conflicting objective functions $f_i : \Re^n \rightarrow \Re$. The *decision vectors* $\vec{x} = (x_1, x_2, \cdots, x_n)^T$ belong to the feasible region $\Omega \subset \Re^n$. We denote the image of the feasible region by $Z \subset \Re^m$ and call it a feasible objective region. The elements of Z are called objective vectors and they consist of m objective (function) values $\vec{f}(\vec{x}) = (f_1(\vec{x}), f_2(\vec{x}), \cdots, f_m(\vec{x}))$. Therefore, each solution $\vec{x}$ provides m different scores (or rewards, or fitness) that are meant to be optimized. Without loss of generality, it is assumed from now on that all objectives are to be maximized.

It is said that a solution $\vec{x}$ *dominates* another solution $\vec{y}$ if and only if:

1) $f_i(\vec{x})$ is not worse than $f_i(\vec{y})$, $\forall i = 1, 2, \ldots, m$.
2) For at least one objective j: $f_j(\vec{x})$ is better than its analogous counterpart in $f_j(\vec{y})$.

When these two conditions apply, it is said that $\vec{x} \preceq \vec{y}$ ($\vec{x}$ dominates $\vec{y}$). The *dominance* condition provides a partial ordering between solutions in the objective space.

However, there are some cases where it cannot be said that $\vec{x} \preceq \vec{y}$ or $\vec{y} \preceq \vec{x}$. In this case, it is said that these solutions are non-dominated with respect to each other. Solutions that are not dominated can be grouped in a *non-dominated set*. Given a non-dominated set P, it is said that P is the *Pareto-set* if there is no other solution in the decision space that dominates any member of P. The corresponding objective vectors of the members in the Pareto-set build a so called *Pareto-front*. The relations between decision and objective space, dominance and the non-dominated set are depicted in Figure 2.

There are many different algorithms in the literature that are proposed to tackle multi-objective optimization problems [17].

Algorithm 1 NSGA-II Algorithm [20]

Input: MOP, N
Output: Non-dominated Set F_0
$t = 0$
$Pop(t) = NewRandomPopulation$
$Q(t) = breed(Pop(t))$ % Generate offspring
while Termination criterion not met **do**
  $U(t) = Pop(t) \cup Q(t)$
  $F = \text{FASTNONDOMINATEDSORT}(U(t))$
  $Pop(t+1) = \emptyset$, $i = 0$
  while $|Pop(t+1)| + |F_i| \leq N$ **do**
    $\text{CROWDINGDISTANCEASSIGNMENT}(F_i)$
    $Pop(t+1) = Pop(t+1) \cup F_i$
    $i = i + 1$
  $\text{SORT}(Pop(t+1))$
  $Pop(t+1) = Pop(t+1) \cup F_i[1 : (N - |Pop(t+1)|)]$
  $Q(t+1) = breed(Pop(t+1))$
  $t = t + 1$
return F_0

A weighted-sum approach is one of the traditional methods in which the objectives are weighted according to user preference. The sum of the weighted objectives builds one objective function to be optimized. The solution to this single-objective problem is one certain solution, ideally on a Pareto-front. By varying the weights, it is possible to converge to different optimal solutions. Such linear scalarization approaches fail to find optimal solutions for problems with non-convex-shape Pareto-fronts [17].

A popular choice for multi-objective optimization problems are evolutionary multi-objective optimization (EMO) algorithms [18], [19]. The goal of EMO algorithms is to find a set of optimal solutions with a good convergence to the Pareto-front as well as a good spread and diversity along the Pareto-front. One of the most well-known algorithms in the literature is the Non-dominated Sorting Evolutionary Algorithm II (NSGA-II) [20], the pseudocode of which is shown in Algorithm 1. As in any evolutionary algorithm, NSGA-II evolves a set of N individuals in a population denoted as Pop with the difference that they are ranked according to a dominance criterion and a crowding distance measure, which are used to maintain a good diversity of solutions. After each iteration of the algorithm, only the best N individuals according to this ranking are maintained to the next generation.

The three main pillars of the NSGA-2 algorithm are:

- Fast non-dominated sorting: The function $F = \text{FASTNONDOMINATEDSORT}(U(t))$ performs a non-dominated sorting and ranks the individuals stored in a set U into several non-dominated fronts denoted as F_i, where the solutions in F_0 are the non-dominated solutions in the entire set. F_i is the set of non-dominated solutions in $R \setminus (F_0 \cup \cdots \cup F_{i-1})$.
- Crowding distance: The function $\text{CROWDINGDISTANCEASSIGNMENT}(F_i)$ assigns to each one of the individuals in F_i a crowding distance value, which is the distance between the individual and its neighbours. The individuals with the smallest

crowding distances are selected with a lower probability than the ones with larger values.
- Elitism: The individuals from the first front F_0 are always ranked first, according to the dominance criterion and their crowding distance. This ensures that the best solutions always survive to the next generation.

For more details about EMO approaches, please refer to [17].

A. Multi-objective Reinforcement Learning (MORL)

Reinforcement Learning (RL) algorithms have also been used with MOPs. RL [21] is a broad field in Machine Learning that studies real-time planning and control situations where an agent has to find the actions (or sequences of actions) that should be used in order to maximize the reward from the environment.

The dynamics of an RL problem are usually captured by a Markov Decision Process (MDP) which is a tuple (S, A, T, R), in which S is the set of possible states in the problem (or game), and s_0 is the initial state. A is the set of available actions the agent can make at any given time, and the transition model $T(s_i, a_i, s_{i+1})$ determines the probability of reaching the state s_{i+1} when action a_i is applied in state s_i. The reward function $R(s_i)$ provides a single value (*reward*) that the agent must optimize, representing the desirability of the state s_i reached. Finally, a policy $\pi(s_i) = a_i$ determines which actions a_i must be chosen from each state $s_i \in S$[1]. One of the most important challenges in RL, as shown in Section II, is the trade-off between exploration and exploitation while trying to act. While learning, a policy must choose between selecting the actions that provided good rewards in the past and exploring new parts of the search space by selecting new actions.

Multi-objective Reinforcement Learning (MORL) [22] changes this definition by using a vector $R = r_1, \ldots, r_m$ as rewards of the problem (instead of a scalar). Thus, MORL problems differ from RL in having more than one objective (here m) that must be maximized. If the objectives are independent or non-conflicting, scalarization approaches such as the weighted sum approach, could be suitable to tackle the problem. Essentially, this would mean using a conventional RL algorithm on a single objective where the global reward is obtained from a weighted-sum of the multiple rewards. However, this is not always the case, as usually the objectives are conflicting and the policy π must balance among them.

Vamplew et al. [22] propose a *single-policy* algorithm which uses a preference order in the objectives (either given by the user or by the nature of the problem). An example of this type of algorithm can be found at [23], where the authors introduce an order of preference in the objectives and constrain the value of the desired rewards. Scalarization approaches would also fit in this category, as shown in the work performed by S. Natarajan et al. [24].

The second type of algorithms, *multiple-policy*, aims to approximate the optimal Pareto-front of the problem. This is the aim of the algorithm proposed in this paper. An example of this type of algorithm is the one given by L. Barrett [25], who proposes the Convex Hull Iteration Algorithm. This algorithm provides the optimal policy for any linear preference function, by learning all policies that define the convex hull of the Pareto-front.

[1]This is a deterministic policy and only valid during acting, not learning

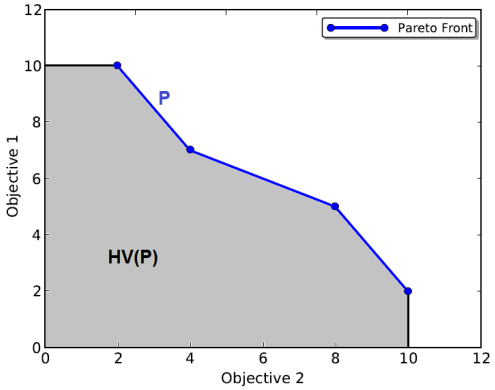


Fig. 3: $HV(P)$ of a given Pareto-front P

B. Metrics

The quality of an obtained non-dominated set can be measured using different diversity or/and convergence metrics [17]. The Hypervolume Indicator (HV) is a popular metric for measuring both the diversity and convergence of non-dominated solutions [26]. This approach can be additionally used to compare different non-dominated sets. Given a Pareto front P, $HV(P)$ is defined as the volume of the objective space dominated by P. More formally, $HV(P) = \mu(x \in \mathbb{R}^d : \exists r \in P \;\; s.t \;\; r \preceq x)$, where μ is the de Lebesgue measure on $\mathbb{R}^d$. If the objectives are to be maximized, the higher the $HV(P)$, the better the front calculated. Figure 3 shows an example of $HV(P)$ where the objective dimension space is $m = 2$.

IV. Multi-Objective Monte Carlo Tree Search

Adapting MCTS into Multi-Objective Monte Carlo Tree Search (MO-MCTS) requires the obvious modification of dealing with multiple rewards instead of just one. As these are collected at the end of a Monte Carlo simulation, the reward value r now becomes a vector $R = r_0, r_1, \ldots, r_m$, where m is the number of objectives to optimize. Derived from this change, the average value $Q(s, a)$ becomes a vector that stores the average reward of each objective. Note that the other statistics ($N(s, a)$ and $N(s)$) do not need to change, as these are just node and action counters. The important question to answer next is how to adapt the vector $Q(s, a)$ to use it in the UCB1 formula (Equation 1).

An initial attempt at Multi-Objective MCTS was addressed by W. Wang and Michele Sebag [27], [28]. In their work, the authors employ a mechanism, based on the HV calculation, to replace the UCB1 equation. The algorithm keeps a Pareto archive (P) with the best solutions found in end game states. Every node in the tree defines $\overline{r}_{sa}$ as a vector of UCB1 values,

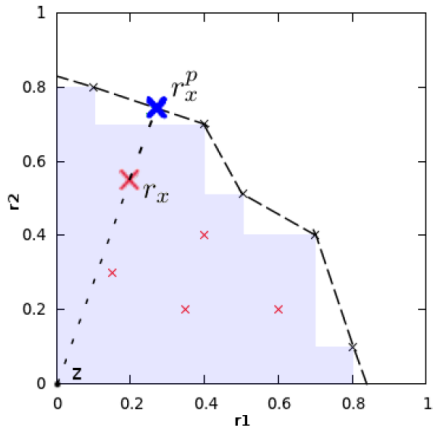


Fig. 4: r_x^p is the projection of the r_x value on the piecewise linear surface (discontinuous line). The shadowed surface represents $HV(P)$. From [27].

in which each $\overline{r}_{sa,i}$ is the result of calculating UCB1 for each particular objective i.

The next step is to define the value for each state and action pair, $V(s,a)$, as in Equation 3. $\overline{r}_{sa}^p$ is the projection of $\overline{r}_{sa}$ into the piecewise linear surface defined by the Pareto archive P (see Figure 4). Then, $HV(P \cup \overline{r}_{sa})$ is declared as the HV of P plus the point $\overline{r}_{sa}$. If $\overline{r}_{sa}$ is dominated by P, the distance between $\overline{r}_{sa}$ and $\overline{r}_{sa}^p$ is subtracted from the HV calculation. The tree policy selects actions based on a maximization of the value of $V(s,a)$.

$$V(s,a) = \begin{cases} HV(P \cup \overline{r}_{sa}) - dist(\overline{r}_{sa}^p, \overline{r}_{sa}) & Otherwise \\ HV(P \cup \overline{r}_{sa}) \quad if \quad \overline{r}_{sa} \preceq P \end{cases} \tag{3}$$

The proposed algorithm was employed successfully in two domains: the DST and the Grid Scheduling problem. In both cases, the results matched the state-of-the-art, albeit at the expense of a high computational cost.

As mentioned before, one of the objectives of this paper is to propose an MO-MCTS algorithm that is suitable for real-time domains. Therefore, an approach different from Wang's is needed, in order to overcome the high computational cost involved in their approach.

In the algorithm proposed in this paper, the reward vector $\overline{r}$ that was obtained at the end of a Monte Carlo simulation is back-propagated through the nodes visited in the last iteration until the root is reached. In the vanilla algorithm, each node would use this vector $\overline{r}$ to update its own accumulated reward vector $\overline{R}$. In the algorithm proposed here, each node in the MO-MCTS algorithm also keeps a local Pareto front (P), updated at each iteration with the reward vector $\overline{r}$ obtained at the end of the simulation. Algorithm 2 describes how the node statistics are updated in MO-MCTS.

Here, if $\overline{r}$ is not dominated by the local Pareto front, it is added to the front and $\overline{r}$ is propagated to its parent. It may also happen that $\overline{r}$ dominates all or some solutions of the local front, in which case the new solution is included and the dominated solutions are removed from the Pareto front. If $\overline{r}$ is dominated by the node's Pareto front, the local Pareto front does not change and there is no need to maintain this propagation up the tree. Three observations can be made about the mechanism described here:

Algorithm 2 Pareto MO-MCTS node update.

```
1: function UPDATE(node, r̄, dominated = false)
2:     node.Visits = node.Visits + 1
3:     node.R̄ = node.R̄ + r̄
4:     if !dominated then
5:         if node.P ⪯ r̄ then
6:             dominated = true
7:         else
8:             node.P = node.P ∪ r̄
9:     UPDATE(node.parent, r̄, dominated)
```

- Each node in the tree has an estimate of the quality of the solutions reachable from there, both as an average (as in the baseline MCTS) and as the best case scenario (by keeping the non-dominated front P).
- By construction, if a reward $\overline{r}$ is dominated by the local front of a node, it is a given that it will be dominated by the nodes above in the tree, so there is no need to update the fronts of the upper nodes, producing little impact on the computational cost of the algorithm.
- It is easy to infer, from the last point, that the Pareto front of a node cannot be worse than the front of its children (in other words, the front of a child will never dominate that of its parent). Therefore, the root of the tree contains the best non-dominated front ever found during the search.

This last detail is important for two main reasons. First of all, the algorithm allows the root to store information as to which action to take in order to converge to any specific solution in the front discovered. This information can be used, when all iterations have been performed, to select the move to perform next. If weights for the different objectives are provided, these weights can be used to select the desired solution in the Pareto front of the root node, and hence select the action that leads to that point. Secondly, the root's Pareto front can be used to measure the global quality of the search using the hypervolume calculation.

Finally, the information stored at each node regarding the local Pareto front can be used to substitute $Q(s,a)$ in the UCB1 equation. The quality of a given pair (s,a) can be obtained by measuring the HV of the Pareto front stored in the node reached from state s after applying action a. This can be defined as $Q(s,a) = HV(P)/N(s)$, and the Upper Confidence Bound equation, referred to here as $MO-UCB$, is described as in Equation 4).

$$a^* = \arg\max_{a \in A(s)} \left\{ HV(P)/N(s) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{4}$$

This algorithm, similar to NSGA-II due to their multi-objective nature, provides a non-dominated front as a solution. However, in planning and control scenarios like the games analyzed in this research, an action must be provided to perform a move in the next step. The question that arises is

how to choose the move to make based on the information available.

As shown before, it is straightforward to discover which actions lead to what points in the front given as a solution: the first gene in the best NSGA-II individual, a root's child in MO-MCTS. Hence, by identifying the point in the Pareto front that the algorithm should converge to, it is possible to execute the action that leads to that point.

In order to select a point in the Pareto front, a weight vector W can be defined, with a dimension m equal to the number of objectives ($W = (w_1, w_2, \ldots, w_m); \sum_i^m w_i = 1$). Two different mechanisms are proposed, for reasons that will be explained in the experiments section (VI):

- Weighted Sum: the action chosen is the one that maximizes the weighted sum of the reward vector multiplied by W, for each point in the front.
- Euclidean distance: the euclidean distance from each point in the Pareto front (normalized in $[0, 1]$) to the vector W is calculated. The action to choose would be the one that leads to the point in the Pareto front with the shortest distance to W.

Note that, in the vanilla MCTS, there is no Pareto front obtained as a solution. Typically, in this case, rewards are calculated as a weighted sum of the objectives and a weight vector W. The action is then chosen following any of the mechanisms usually employed in the literature: the action taken more often from the root; the one that leads to the best reward found; the move with the highest expected reward; or the action that maximizes Equation 1 in the root.

V. Benchmarks

Two different games are used in this research to analyze the performance of the algorithm proposed.

A. Deep Sea Treasure

The Deep Sea Treasure (DST) is a popular multi-objective problem introduced by Vamplew et al. [22]. In this single-player puzzle, the agent controls a submarine with the objective of finding a treasure located at the bottom of the sea. The world is divided into a grid of 10 rows and 11 columns, and the vessel starts at the top left board position. There are three types of cells: empty cells (or water), that the submarine can traverse; ground cells that, as the edges of the grid, cannot be traversed; and treasure cells, that provide different rewards and finish the game. Figure 5 shows the DST environment.

The ship can perform four different moves: *up*, *down*, *right* and *left*. If the action applied takes the ship off the grid or into the sea floor, the vessel's position will not change. There are two objectives in this game: the number of moves performed by the ship, which must be minimized, and the value of the treasure found, which should be maximized. As can be seen in Figure 5, the most valuable treasures are at a greater distance from the initial position, so the objectives are in conflict.

Additionally, the agent can only make up to 100 plies or moves. This allows the problem to be defined as the maximization of two rewards: $(\rho_p, \rho_v) = (100 - plies, treasureValue)$. Should the ship perform all moves without reaching a treasure, the result would be (0,0). At each step, the score of a location with no treasure is $(-1, 0)$.

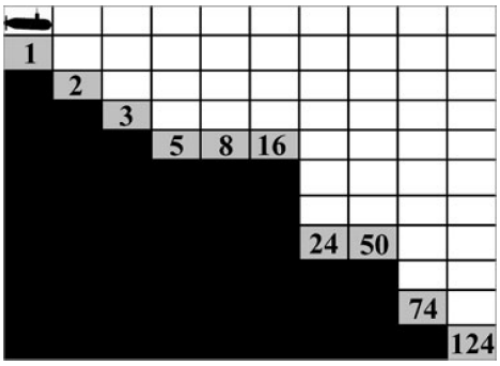


Fig. 5: Environment of the Deep Sea Treasure (from [22]): grey squares represent the treasure (with their different values) available in the map. The black cells are the sea floor and the white ones are the positions that the vessel can occupy freely. The game ends when the submarine picks one of the treasures.

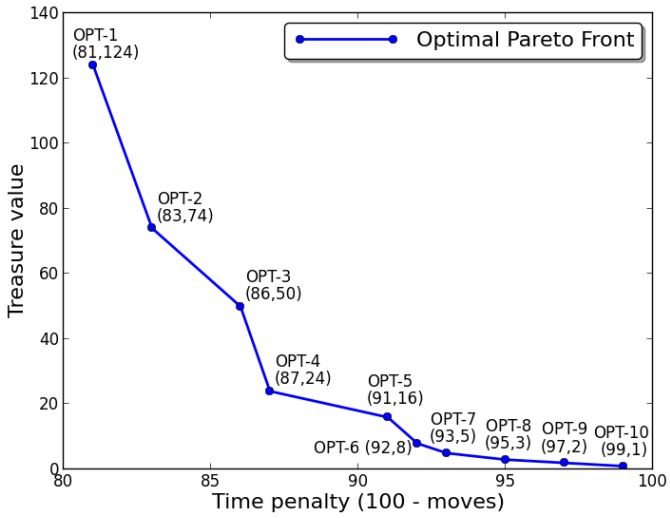


Fig. 6: Optimal Pareto Front of the Deep Sea Treasure, with both objectives to be maximized.

The optimal Pareto front of the DST is shown in Figure 6. There are 10 non-dominated solutions in this front, one per each treasure in the board. The front is globally concave, with local concavities at the second $(83, 74)$, fourth $(87, 24)$ and sixth $(92, 8)$ points from the left. The HV value of the optimal front is 10455.

Section III introduced the problems of linear scalarization approaches when facing non-convex optimal Pareto fronts. The concave shape of the DST's optimal front means that those approximations converge to the non dominated solutions located at the edges of the Pareto front: $(81, 124)$ and $(99, 1)$. Note that this happens independently from the weights chosen for the linear approximation: some solutions of the front just can't be reached with these approaches. Thus, successful approaches should be able to find all elements of the optimal Pareto front and converge to any of the non dominated solutions.

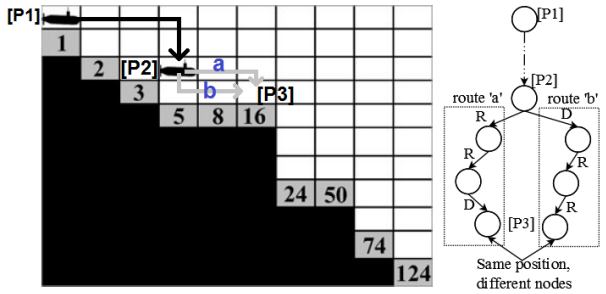


Fig. 7: Example of two different sequences of actions (R: Right, D: Down) that lie in the same position in the map, but at a different node in the tree.

B. Heuristics for DST

The DST is a problem especially suited for the use of Transposition Tables (TT) [29] within MCTS. TT is a technique used to optimize tree search-based algorithms when two or more states can be found at different locations in the tree. It consists of sharing information between these equivalent states in a centralized manner, in order to avoid managing these positions as completely different states. Figure 7 shows an example of this situation in the DST.

In this example, the submarine starts in the initial position ($P1$, root of the tree) and makes a sequence of moves that places it in $P2$. From this location, two optimal trajectories to move to $P3$ would be route *a* and *b*. In a tree that does not use TT, there would be two different nodes to represent $P3$, although the location and number of moves performed up to this point are the same (thus, the states are equivalent). It is worthwhile highlighting that the coordinates of the vessel are not enough to identify two equivalent states, the number of moves is also needed: imagine a third route from $P2$ to $P3$ with the moves: *Up*, *Right*, *Right*, *Down*, *Down*. As the submarine performs 5 moves, the states are not the same, and the node where the ship is in $P3$ now would be two levels deeper in the tree.

TT tables are implemented in the MCTS algorithms tested in this benchmark by using hash tables that store a *representative* node for each pair $(position, moves)$ found. The key of the hash map then needs to be obtained from three values: position coordinates x and y of the ship in the board, and number of moves, indicated by the depth of the node in the tree. Hence, transpositions can only happen at the same depth within the tree, a feature that has been successfully tried before in the literature [30].

As DST has two different objectives, number of moves and value of the treasure, the quality of a state can be assessed by two rewards, ρ_p and ρ_v, for each objective respectively. ρ_p is adjusted to be maximized using the maximum number of moves in the game, while ρ_v is simply the value of the cell that holds the treasure. Therefore, the reward vector to maximize is defined as $\overline{r} = \{\rho_p, \rho_v\}$. Equation 5 summarizes these rewards:

$$\begin{aligned} \rho_p &= 100 - moves \\ \rho_v &= treasureValue \end{aligned} \tag{5}$$

C. Multi-Objective PTSP

The Multi-Objective Physical Travelling Salesman Problem (MO-PTSP) is a game that was employed in a competition held at the IEEE Conference on Computational Intelligence in Games (CIG) in 2013. This was a modification of the Physical Travelling Salesman Problem (PTSP), previously introduced by Perez et al. [31]. The MO-PTSP is a real-time game where the agent navigates a ship and must visit 10 waypoints scattered around the maze. All waypoints must be visited to consider a game as complete, and a game-tick counter is reset every time a waypoint is collected, finishing the game prematurely (and unsuccessfully) if 800 game steps are reached before visiting another waypoint.

This needs to be accomplished while optimizing three different objectives: 1) **Time**: the player must collect all waypoints scattered around the maze in as few time steps as possible; 2) **Fuel**: the fuel consumption by the end of the game must be minimized; and 3) **Damage**: the ship should end the game with as little damage as possible.

In the game, the agent must provide an action every 40 milliseconds. The available actions are combinations of two different inputs: *throttle* (that could be *on* or *off*) and *steering* (that could be *straight*, *left* or *right*). This allows for 6 different actions that modify the ship's position, velocity and orientation. These vectors are kept from one step to the next, keeping the inertia of the ship, and making the navigation task not trivial.

The ship starts with 5000 units of fuel, and one unit is spent every time an action supplied has the throttle input *on*. There are, however, two ways of collecting fuel: each waypoint visited grants the ship 50 more units of fuel; and there are also fuel canisters scattered around the maze that provide 250 more units.

Regarding the third objective, the ship can suffer damage in two different ways: by colliding with obstacles and driving through lava. In the former case, the ship can collide with normal obstacles (that subtract 10 units of damage) and especially damaging obstacles (30 units). In the latter, lava lakes are abundant in MO-PTSP levels and, in contrast with normal surfaces, they deal one unit of damage for each time step the ship spends over this type of surface. All these subtractions are deducted from an initial counter of 5000 points of damage.

Figure 8 shows an example of an MO-PTSP map, as drawn by the framework. Waypoints not yet visited are painted as blue circles, while those already collected are depicted as empty circles. The green ellipses represent fuel canisters, normal surfaces are drawn in brown and lava lakes are printed as red-dotted yellow surfaces. Normal obstacles are black and damaging obstacles are drawn in red. Blue obstacles are elastic walls that produce no damage to the ship. The vessel is drawn as a blue polygon and its trajectory is traced with a black line.

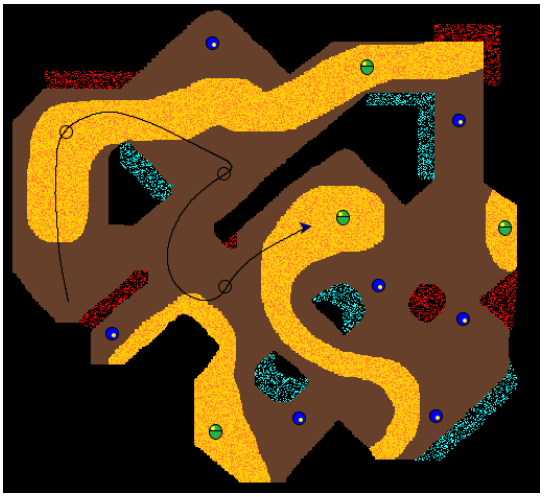


Fig. 8: Sample MO-PTSP map.

D. Heuristics for MO-PTSP

All the algorithms tested in the MO-PTSP in this research employ macro-actions, a concept that can be used for coarsening the action space by applying the action chosen by the control algorithm in several consecutive cycles, instead of just in the next one.

Previous research in PTSP [14], [32] suggests that using macro-actions for real-time navigation domains increases the performance of the algorithms, and it has been used in the previous PTSP competitions by the winner and other entries.

A macro-action of length L is defined as a repetition of a given action during L consecutive time steps. The main advantage of macro-actions is that the control algorithms can see further in the future, and then reduce the implication of open-endedness in real-time games (see Section II). As this reduces the search space significantly, a better algorithm performance is allowed. A consequence of using macro-actions is also that the algorithm can employ L consecutive steps to plan the next move (the next macro-action) to make. Therefore, instead of spending 40 milliseconds, as in MO-PTSP, to define the next action, the algorithm can employ $40 \times L$ milliseconds for this task, and hence perform a more extensive search.

In MO-PTSP, the macro-action size is $L = 15$, a value that has shown its proficiency before in PTSP. For more information about macro-actions and their application to real-time navigation problems such as the PTSP, the reader is referred to [14].

In order to evaluate a game state in MO-PTSP, three different measures or rewards are taken, ρ_t, ρ_f, ρ_d, one for each one of the objectives: time, fuel and damage, respectively. All these rewards are defined so they have to be maximized. The first reward uses a measure of distance for the time objective, as indicated in Equation 6:

$$\rho_t = 1 - d_t/d_M \tag{6}$$

The MO-PTSP framework includes a path-finding library that allows controllers to query the shortest distance of a route of waypoints. d_t indicates the distance from the current position until the last waypoint, following the desired route, and d_M is the distance of the whole route from the starting position. Minimizing the distance to the last waypoint will lead to the end of the game.

Equation 7 shows how the value of the fuel objective, ρ_f, is obtained:

$$\rho_f = (1 - (\lambda_t/\lambda_0)) \times \alpha + \rho_t \times (1 - \alpha) \tag{7}$$

λ_t is the fuel consumed so far, and λ_0 is the initial fuel at the start of the game. α is a value that balances between the fuel component and the time objective from Equation 6. Note that, as waypoints need to keep being visited, it is necessary to include a distance measure for this reward. Otherwise, an approach that prioritizes this objective would not minimize distance to waypoints at all (the ship could just stand still: no fuel consumed is optimal), and therefore would not complete the game. The value of α has been determined empirically, in order to provide a good balance between these two components, and is set to 0.66.

Finally, Equation 8 gives the method used to calculate the damage objective, ρ_d:

$$\rho_d = \begin{cases} (1 - (g_t/g_M)) \times \beta_1 + \rho_t \times (1 - \beta_1), & sp > \gamma \\ (1 - (g_t/g_M)) \times \beta_2 + \rho_t \times (1 - \beta_2), & sp \leq \gamma \end{cases} \tag{8}$$

g_t is the damage suffered so far in the game, and g_M is the maximum damage the ship can take. In this case, three different variables are used to regulate the behaviour of this objective: γ, β_1 and β_2. Both β_1 and β_2 have the same role as α in Equation 7: they balance between the time objective and the damage measure. The difference is that β_1 is used in high speeds, while β_2 is employed with low velocities. This is distinguished by the parameter γ, that can be seen as a threshold value for the ship's speed (sp). This differentiation is made in order to avoid low speeds in lava lakes, as this significantly increases the damage suffered. The values for these variables have also been determined empirically, and they are set to $\gamma = 0.8$, $\beta_1 = 0.75$ and $\beta_2 = 0.25$.

The final reward vector to be maximized is therefore $\overline{r} = \{\rho_t, \rho_f, \rho_d\}$. It is important to highlight again that these three rewards are the same for all the algorithms tested in the experiments.

VI. Experimentation

The experiments performed in this research compare three different algorithms in the two benchmarks presented in Section V: a single objective MCTS (referred to here simply as *MCTS*), the Multi-Objective MCTS (*MO-MCTS*) and a rolling horizon version of the NSGA-II algorithm described in Section III (*NSGA-II*). This NSGA-II version evolves a population where the individuals are sequences of actions (macro-actions in the MO-PTSP case), obtaining the fitness from the state of the game after applying the sequence. The population sizes, determined empirically, were set to 20 and 50 individuals for DST and MO-PTSP respectively. The value of C in Equation 1 and 4 is set to $\sqrt{2}$.

All the algorithms have a limited number of evaluations before providing an action to perform. In order for these games to be real-time, the time budget allowed is close to 40 milliseconds. With the objective of avoiding congestion peaks at the machine where the experiments are run, the average number of evaluations possible in 40 ms is calculated and employed in the tests. This leads to 4500 evaluations in the DST, and 500 evaluations for MO-PTSP, using the same server where the PTSP and MO-PTSP competitions were run[2].

A. Results in DST

As the optimal Pareto front of the DST is known, a measure of performance can be obtained by observing the percentage of times these solutions are found by the players. As the solution that the algorithms converge to depends on the weights vector employed during the search (see end of Section IV), the approach taken here is to provide different weight vectors W and analyze them separately.

The weight vector for DST has two dimensions. This vector is here referred to as $W = (w_p, w_v)$, where w_p weights moves; and w_v weights the treasure value ($w_v = 1 - w_p$). w_p takes values between 0 and 1, with a precision of 0.01, and 100 runs have been performed for each pair (w_p, w_v). Hence, the game has been played a total of 10000 times, for each algorithm.

Figure 9 shows the results obtained after these experiments were performed. The first point to note is that MCTS only converges (mostly) to the two optimal points located at the edges of the optimal Pareto front (*OPT-1* and *OPT-10*, see also Figure 6). This is an expected result, as K. Deb. suggested in [17] and was also discussed before in Section III: linear scalarization approaches only converge to the edges of the optimal Pareto front if its shape is non-convex.

The results show clearly how approximating the optimal Pareto front allows for finding all possible solutions. Both the NSGA-II and MO-MCTS approaches are able to converge to any solution in the front given the appropriate weight vector. It is important to highlight that these two algorithms employed the Euclidean distance action selection (see Section IV). Other experiments, not included in this paper, showed that weighted sum action selection provides similar results to MCTS (that is, convergence to the edges of the front). The crucial distinction to make here is that both algorithms, NSGA-II and MO-MCTS, allow for better action selection mechanisms, which are able to overcome this problem by approximating a global Pareto front.

Finally, in the comparison between NSGA-II and MO-MCTS, the latter algorithm obtains higher percentages for each one of the points in the front. This result suggests that, with a limited number of iterations/evaluations, the proposed algorithm is able to explore the search space more efficiently than a rolling horizon version of a state-of-the-art NSGA-II.

B. Results in MO-PTSP

The same three algorithms have been tested in the MO-PTSP domain. The experiments are performed in the 10 maps available within the framework [3]. In this case, the optimal Pareto front is not known in advance, and normally differs from one map to another. Hence, the mechanisms to compare the performance of the algorithms tested need to be different than the one used for DST.

Map	$W : (w_t, w_f, w_d)$	Time	Fuel	Damage
Map 1	$(0.33, 0.33, 0.33)$	1654 ± 7	131 ± 2	846 ± 13
	$(0.1, 0.3, 0.6)$	1657 ± 8	130 ± 2	$\mathbf{773 \pm 11}$
	$(0.1, 0.6, 0.3)$	1681 ± 11	131 ± 2	837 ± 15
	$(0.6, 0.1, 0.3)$	1649 ± 8	132 ± 2	833 ± 13
Map 2	$(0.33, 0.33, 0.33)$	1409 ± 7	235 ± 4	364 ± 3
	$(0.1, 0.3, 0.6)$	1402 ± 6	236 ± 5	$\mathbf{354 \pm 2}$
	$(0.1, 0.6, 0.3)$	1416 ± 8	$\mathbf{219 \pm 4}$	360 ± 3
	$(0.6, 0.1, 0.3)$	1396 ± 8	245 ± 5	361 ± 2
Map 3	$(0.33, 0.33, 0.33)$	1373 ± 6	221 ± 3	301 ± 7
	$(0.1, 0.3, 0.6)$	1378 ± 5	211 ± 3	$\mathbf{268 \pm 5}$
	$(0.1, 0.6, 0.3)$	1385 ± 6	$\mathbf{203 \pm 4}$	291 ± 7
	$(0.6, 0.1, 0.3)$	$\mathbf{1363 \pm 4}$	229 ± 4	285 ± 7
Map 4	$(0.33, 0.33, 0.33)$	1383 ± 6	291 ± 5	565 ± 5
	$(0.1, 0.3, 0.6)$	1385 ± 7	304 ± 4	$\mathbf{542 \pm 4}$
	$(0.1, 0.6, 0.3)$	1423 ± 6	$\mathbf{273 \pm 4}$	583 ± 5
	$(0.6, 0.1, 0.3)$	1388 ± 6	309 ± 4	559 ± 5
Map 5	$(0.33, 0.33, 0.33)$	1405 ± 7	467 ± 4	559 ± 4
	$(0.1, 0.3, 0.6)$	1431 ± 9	447 ± 4	541 ± 4
	$(0.1, 0.6, 0.3)$	1467 ± 9	$\mathbf{411 \pm 5}$	567 ± 5
	$(0.6, 0.1, 0.3)$	1399 ± 9	469 ± 4	547 ± 3
Map 6	$(0.33, 0.33, 0.33)$	1575 ± 7	549 ± 5	303 ± 4
	$(0.1, 0.3, 0.6)$	1626 ± 9	540 ± 6	286 ± 5
	$(0.1, 0.6, 0.3)$	1703 ± 11	$\mathbf{499 \pm 4}$	316 ± 7
	$(0.6, 0.1, 0.3)$	1571 ± 7	559 ± 5	294 ± 4
Map 7	$(0.33, 0.33, 0.33)$	1434 ± 5	599 ± 6	284 ± 6
	$(0.1, 0.3, 0.6)$	1475 ± 10	602 ± 5	$\mathbf{243 \pm 6}$
	$(0.1, 0.6, 0.3)$	1489 ± 12	$\mathbf{549 \pm 3}$	264 ± 6
	$(0.6, 0.1, 0.3)$	$\mathbf{1407 \pm 8}$	618 ± 5	270 ± 6
Map 8	$(0.33, 0.33, 0.33)$	1761 ± 9	254 ± 5	382 ± 3
	$(0.1, 0.3, 0.6)$	1804 ± 10	269 ± 4	$\mathbf{357 \pm 4}$
	$(0.1, 0.6, 0.3)$	1826 ± 10	$\mathbf{230 \pm 3}$	392 ± 7
	$(0.6, 0.1, 0.3)$	$\mathbf{1732 \pm 9}$	311 ± 8	379 ± 6
Map 9	$(0.33, 0.33, 0.33)$	2501 ± 14	926 ± 6	574 ± 9
	$(0.1, 0.3, 0.6)$	2503 ± 10	921 ± 10	$\mathbf{524 \pm 8}$
	$(0.1, 0.6, 0.3)$	2641 ± 14	$\mathbf{833 \pm 5}$	574 ± 14
	$(0.6, 0.1, 0.3)$	$\mathbf{2470 \pm 9}$	956 ± 5	573 ± 8
Map 10	$(0.33, 0.33, 0.33)$	1430 ± 8	630 ± 4	205 ± 2
	$(0.1, 0.3, 0.6)$	1493 ± 13	615 ± 4	209 ± 2
	$(0.1, 0.6, 0.3)$	1542 ± 10	$\mathbf{554 \pm 4}$	229 ± 5
	$(0.6, 0.1, 0.3)$	$\mathbf{1378 \pm 5}$	663 ± 6	202 ± 4

TABLE I: MO-MCTS results in MO-PTSP with different weight vectors. Values in bold indicate the best results on each map, when they are separated at least by 2 standard errors from the others.

The idea is as follows. First of all, 4 different weight vectors are tested: $W_1 = (0.33, 0.33, 0.33)$, $W_2 = (0.1, 0.3, 0.6)$, $W_3 = (0.1, 0.6, 0.3)$ and $W_4 = (0.6, 0.1, 0.3)$, where each w_i corresponds to the weight for an objective (w_t for time, w_f for fuel and w_d for damage, in this order). W_1 treats all objectives as having the same weight, while the other three give more relevance to different objectives. These vectors then provide a wide spectrum for weights in this benchmark. In this case, MO-MCTS uses these weights to select an action based on a weighted sum, which in preliminary experiments has shown better performance than the Euclidean distance mechanism.

The first point to check is if the different weight vectors affect the solutions obtained by MO-MCTS. Table I shows the

[2]Intel Core i5, 2.90GHz, 4GB of RAM.

[3]Available at www.ptsp-game.net

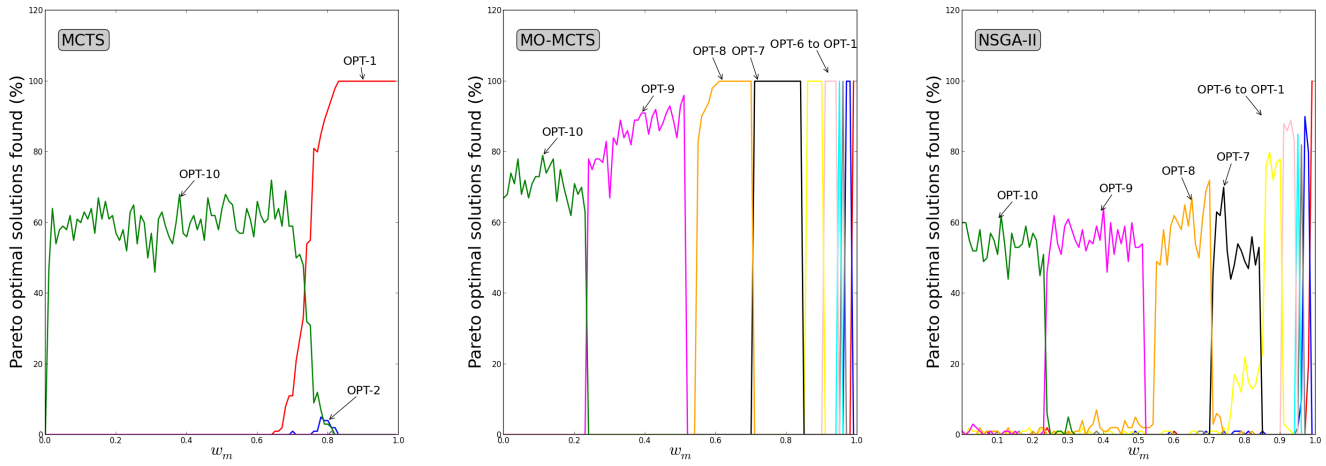


Fig. 9: Results in DST: percentages of each optima found during 100 games played with different weight vectors. Scalarization approaches converge to the edges of the optimal front, whereas Pareto approaches are able to find all optimal solutions. The proposed algorithm, MO-MCTS, finds these solutions significantly more often than NSGA-II.

results of executing the MO-MCTS controller during 30 games in each one of the 10 maps of the MO-PTSP, for every weight vector. It can be seen that the highest weight in W leads, in most of the cases, to the best solution in that objective in the map.

Some exceptions can be explained by analysing the specific maps: in map 1, a fuel canister is always collected near the end, restoring the fuel level to its maximum. This leaves too few cycles to make a difference in the controller (note also that this is the map with the smallest fuel consumption). Also, in map 10, there is no difference in the damage objective, however map 10 is a map with no obstacles to damage the ship (thus only lava lakes deal damage). This also results in this map being the one with the lowest overall damage. It can also be seen that, in those maps where time has priority, the results for this objective are not as dominant as the other two. This can be explained by the fact that the time heuristic is actually part of the fuel and damage heuristics (Equation 6 in Eqs. 7 and 8).

These results suggest that the weights effectively work on making the algorithm converge to different points in the Pareto front discovered by the algorithm. Now, it is time to compare these results with the other algorithms. In order to do this, the same number of runs is performed for NSGA-II and MCTS in the 10 maps of the benchmark.

In order to compare the several algorithms, the results are examined in pairs, in terms of dominance. The procedure is as follows: once all games on a single map have been run, the Mann-Whitney-Wilcoxon non-parametric test with 95% confidence is calculated on the three objectives. If the measures on all objectives are assumed to be drawn from different distributions, their averages are compared for a dominance test (if not, then no dominance relationship can be derived from the results). If one result dominates another, then the algorithm dominates the other in that particular map.

Extending this comparison to all maps, each pair of algorithms ends with a triplet $(D, \emptyset, d)$. D is the number of maps where the first algorithm dominates the second. $\emptyset$ is the amount of maps where no dominance can be established, either because the Mann-Whitney-Wilcoxon non-parametric test failed, or because there is no dominance according to the dominance rules described in Section III. Finally, d states the number of maps where the first is dominated by the second. For example, a triplet $(D, \emptyset, d) = (8, 2, 0)$ comparing algorithms A and B would mean that the results obtained by A dominate those from B in 8 of the 10 maps, and that it is not possible to derive any dominance in the other 2. Table II summarizes these results for all the algorithms tested.

One of the first things to notice is that MO-MCTS dominates MCTS and NSGA-II in most of the maps, and it is never dominated in any. In particular, the dominance of MO-MCTS over MCTS is outstanding, even dominating in all 10 maps for two of the weight vectors. MO-MCTS also dominates NSGA-II in more maps than in those where there is no dominance, and is never dominated by NSGA-II in any map.

It is also interesting to see that NSGA-II dominates the weighted sum version of MCTS, although for the vector W_2 there is a technical draw, as they dominate each other in 4 different maps each and there is no dominance in the other 2.

Additionally, Table II contains a fourth entry, `PurofMovio`, that the algorithms are compared against. `PurofMovio` was the winning entry of the 2013 MO-PTSP competition, a controller based on a weighted-sum MCTS approach (see [16] for details of its implementation). As can be seen, `PurofMovio` obtains better results than the algorithm proposed in this paper.

However, it is very important to highlight that `PurofMovio` is not using the same heuristics as the ones presented in this research. Hence, nothing can be concluded from making pairwise comparisons directly with the winning entry of the competition. It is likely that `PurofMovio`'s heuristics are more efficient than the ones presented here, but the goal of this paper is not to develop the best heuristics for the MO-PTSP, but to provide an insight into how a multi-objective version of MCTS compares to other algorithms using the same heuristics.

	$W : (w_t, w_f, w_d)$	**MO-MCTS** $(D, \emptyset, d)$	**MCTS** $(D, \emptyset, d)$	**NSGA-II** $(D, \emptyset, d)$	**PurofMovio** $(D, \emptyset, d)$
MO-MCTS	$W_1 : (0.33, 0.33, 0.33)$	–	$(8, 2, 0)$	$(8, 2, 0)$	$(0, 5, 5)$
	$W_2 : (0.1, 0.3, 0.6)$		$(10, 0, 0)$	$(4, 6, 0)$	$(0, 6, 4)$
	$W_3 : (0.1, 0.6, 0.3)$		$(8, 2, 0)$	$(7, 3, 0)$	$(0, 5, 5)$
	$W_4 : (0.6, 0.1, 0.3)$		$(10, 0, 0)$	$(3, 7, 0)$	$(0, 3, 7)$
MCTS	$W_1 : (0.33, 0.33, 0.33)$	$(0, 8, 2)$	–	$(0, 2, 8)$	$(0, 2, 8)$
	$W_2 : (0.1, 0.3, 0.6)$	$(0, 0, 10)$		$(4, 2, 4)$	$(0, 3, 7)$
	$W_3 : (0.1, 0.6, 0.3)$	$(0, 2, 8)$		$(0, 1, 9)$	$(0, 6, 4)$
	$W_4 : (0.6, 0.1, 0.3)$	$(0, 0, 10)$		$(3, 3, 4)$	$(0, 1, 9)$
NSGA-II	$W_1 : (0.33, 0.33, 0.33)$	$(0, 2, 8)$	$(8, 2, 0)$	–	$(0, 4, 6)$
	$W_2 : (0.1, 0.3, 0.6)$	$(0, 6, 4)$	$(4, 2, 4)$		$(0, 4, 6)$
	$W_3 : (0.1, 0.6, 0.3)$	$(0, 3, 7)$	$(9, 1, 0)$		$(0, 5, 5)$
	$W_4 : (0.6, 0.1, 0.3)$	$(0, 7, 3)$	$(4, 3, 3)$		$(0, 4, 6)$
PurofMovio	$W_1 : (0.33, 0.33, 0.33)$	$(5, 5, 0)$	$(8, 2, 0)$	$(6, 4, 0)$	–
	$W_2 : (0.1, 0.6, 0.3)$	$(4, 6, 0)$	$(7, 3, 0)$	$(6, 4, 0)$	
	$W_3 : (0.1, 0.3, 0.6)$	$(5, 5, 0)$	$(6, 4, 0)$	$(5, 5, 0)$	
	$W_4 : (0.6, 0.1, 0.3)$	$(7, 3, 0)$	$(9, 1, 0)$	$(6, 4, 0)$	

TABLE II: Results in MO-PTSP: Each cell indicates the triplet $(D, \emptyset, d)$, where D is the number of maps where the row algorithm dominates the column one, $\emptyset$ is the amount of maps where no dominance can be established, and d states the number of maps where the row algorithm is dominated by the column one. All the algorithms followed the same route (order of waypoints and fuel canisters) in every map tested.

Nonetheless, the inclusion of `PurofMovio` in this comparison is not pointless: it is possible to assess the quality of the three algorithms tested here by comparing their performance relatively, against this high quality entry. Attending to this criteria, it can be seen how MO-MCTS is the algorithm that is dominated less often by `PurofMovio`, producing similar results on an average of 4.75 out of the 10 maps, and being dominated in 5.25 maps. MCTS and NSGA-II are dominated more often than MO-MCTS, being dominated in an average of 7.5 and 5.75 of the maps, respectively. This comparative result shows again that MO-MCTS is achieving the best results among the three algorithms compared here.

C. A step further in MO-PTSP: segments and weights

There is another aspect that can be further improved in the MO-PTSP benchmark, and is also applicable to other domains. It is naive to think that a unique weight vector will be the ideal one for the whole game. Specifically in the MO-PTSP, there are regions of the map where there are more obstacles or lava lakes, hence the ship is most likely to suffer higher damage there. Also, the route followed during the game affects the relative ideal speed between waypoints, or perhaps a fuel canister may be picked up, which will affect how the fuel objective will be managed. In general, many real-time games go through different phases, with different objectives and priorities.

A way to provide different weights at different times in MO-PTSP is straightforward. Given the route of waypoints (and fuel canisters) being followed, one can divide it into *segments*, where each segment starts and ends with a waypoint (or fuel canister). Then, each segment can be assigned a particular weight vector W.

The question is then how to assign these weight vectors. Three different ways can be devised:

- Manually set the weight vectors. This was attempted and it proved to be a non trivial task.
- Setting the appropriate weight for each segment dynamically, based on the segment's characteristics. This involves the creation or discovery of features and some kind of function approximation to assign the values.
- Learn, for each specific map, the combination of weight vectors that produces better results.

This section shows some initial results obtained when testing the third variant, using a stochastic hill climbing algorithm on each map. The goal is to check if, by varying the weight vectors between segments, better solutions can be achieved.

An individual is identified by a string of integers, where each integer refers to one of the weight vectors utilized in the previous sections ($1 = W_1 = (0.33, 0.33, 0.33)$, $2 = W_2 = (0.1, 0.3, 0.6)$ and $3 = W_3 = (0.1, 0.6, 0.3)$. W_4 has been left out of this experiment, as it has shown to be the least influential weight vector). The solution is evaluated playing a particular map 10 times, and its fitness is obtained by calculating the average of those runs. A population of 10 individuals is kept, and the solutions of the initial population are created either randomly, or mutated from base individuals. These base individuals all have segments with the same weight vector W_1, W_2, or W_3.

The best solution, determined by dominance, is kept and promoted to the next generation, where it is mutated to generate other individuals of the population. Also, a portion of the individuals of the next population is created uniformly at random at every generation, until the end of the algorithm, which is established at 50 generations.

Table III shows the results obtained on each run, one per map. Each row corresponds to a run in the associated map, and it provides different results depending on the weights vector. The top three are the base individuals, taken from Table II for comparison. The forth result on each row is the best one after the run. Each genome is a string of the form $abc \dots z$, that represents $W_a W_b W_c \dots W_z$, where each element is a weight vector used in that particular segment. The last column indicates if the evolved individual dominates ($\preceq$) or not ($\emptyset$) each one of the base genomes for that particular map.

The results suggest some interesting ideas. First of all, it is indeed possible to obtain better results by varying the

Map	Weight genome	Time	Fuel	Damage	D
Map 1	11111111111111	1654 ± 7	131 ± 2	846 ± 13	$\preceq \preceq \preceq$
	22222222222222	1657 ± 8	130 ± 2	773 ± 11	
	33333333333333	1681 ± 11	131 ± 2	837 ± 15	
	32312212331112	1619 ± 12	130 ± 2	744 ± 15	
Map 2	11111111111111	1409 ± 7	235 ± 4	364 ± 3	$\preceq \preceq \preceq$
	22222222222222	1402 ± 6	236 ± 5	354 ± 2	
	33333333333333	1416 ± 8	219 ± 4	360 ± 3	
	23131312323213	1390 ± 10	210 ± 3	353 ± 3	
Map 3	11111111111111	1373 ± 6	221 ± 3	301 ± 7	$\preceq \preceq \emptyset$
	22222222222222	1378 ± 5	211 ± 3	268 ± 5	
	33333333333333	1385 ± 6	203 ± 4	291 ± 7	
	11122222112231	1358 ± 9	219 ± 7	263 ± 12	
Map 4	11111111111111	1383 ± 6	291 ± 5	565 ± 5	$\preceq \preceq \emptyset$
	22222222222222	1385 ± 7	304 ± 4	542 ± 4	
	33333333333333	1423 ± 6	273 ± 4	583 ± 5	
	11121131212112	1360 ± 4	282 ± 5	540 ± 4	
Map 5	11111111111111	1405 ± 7	467 ± 4	559 ± 4	$\preceq \preceq \emptyset$
	22222222222222	1431 ± 9	447 ± 4	541 ± 4	
	33333333333333	1467 ± 9	411 ± 5	567 ± 5	
	21311213111211	1397 ± 11	448 ± 10	535 ± 5	
Map 6	11111111111111	1575 ± 7	549 ± 5	303 ± 4	$\preceq \preceq \emptyset$
	22222222222222	1626 ± 9	540 ± 6	286 ± 5	
	33333333333333	1703 ± 11	499 ± 4	316 ± 7	
	31121312111111	1570 ± 16	535 ± 10	266 ± 6	
Map 7	11111111111111	1434 ± 5	599 ± 6	284 ± 6	$\preceq \preceq \emptyset$
	22222222222222	1475 ± 10	602 ± 5	243 ± 6	
	33333333333333	1489 ± 12	549 ± 3	264 ± 6	
	11332211321332	1401 ± 12	563 ± 4	230 ± 12	
Map 8	11111111111111	1761 ± 9	254 ± 5	382 ± 3	$\preceq \preceq \emptyset$
	22222222222222	1804 ± 10	269 ± 4	357 ± 4	
	33333333333333	1826 ± 10	230 ± 3	392 ± 7	
	23221131313323	1747 ± 11	247 ± 9	363 ± 9	
Map 9	11111111111111	2501 ± 14	926 ± 6	574 ± 9	$\preceq \preceq \emptyset$
	22222222222222	2503 ± 10	921 ± 10	524 ± 8	
	33333333333333	2641 ± 14	833 ± 5	574 ± 14	
	21132331333223	2463 ± 19	891 ± 8	523 ± 9	
Map 10	11111111111111	1430 ± 8	630 ± 4	205 ± 2	$\preceq \preceq \emptyset$
	22222222222222	1493 ± 13	615 ± 4	209 ± 2	
	33333333333333	1542 ± 10	554 ± 4	229 ± 5	
	11311111322231	1418 ± 9	623 ± 9	197 ± 2	

TABLE III: MO-PTSP Results with different weights. The last column indicates if the evolved individual dominates ($\preceq$) or not ($\emptyset$) each one of the base genomes for that particular map.

weights of the objectives along the route: in 22 out of the 30 comparisons made, the evolved solution dominates the base individuals and, in the other 8, both solutions in the comparison would be in the same Pareto front of solutions.

If the focus is set on what particular weights are more successful, it is interesting to see that the best configurations found are better, in all maps, than the base individuals with all weights equal to W_1 ($w_t = 0.33, w_f = 0.33, w_d = 0.33$) and W_2 ($w_t = 0.1, w_f = 0.3, w_d = 0.6$). In other words, it was always possible to find a combination of weight vectors that performed better than approaches that gave the same weight to all objectives, and also better than the ones that prioritize low damage.

It is also worthwhile mentioning that in those cases where dominance over the base individual was not achieved, the base was the one that prioritized fuel. Actually, it can be seen that in these cases, the result in the fuel objective is the one that prevents the dominance from happening, obtaining a better value than the evolved solution in this particular objective.

VII. Conclusions and Future Work

This paper presents a multi-objective version of Monte Carlo Tree Search (MO-MCTS) for real-time games: scenarios where the time budget allowed for finding out the best possible action to apply next is close to 40ms. MO-MCTS is tested, in comparison with a single-objective MCTS algorithm and a rolling horizon NSGA-II, in two different real-time games, the Deep Sea Treasure (DST) and the Multi-Objective Physical Travelling Salesman Problem (MO-PTSP). This comparison is made by using the same heuristics that determine the value of a given state, so the focus is set on how the algorithms explore the search space on each problem.

The results obtained in this study show the strength of the algorithm proposed, as it provides better solutions than the ones obtained by the other algorithms in comparison. Also, the MO-MCTS approach samples successfully across the different solutions in the Pareto front (optimal front in the DST, the best found in MO-PTSP), depending on the weights provided for each objective. Finally, some initial results are obtained applying the idea of using distinct weight vectors for the objectives in different situations within the game, showing that it is possible to improve the performance of the algorithms.

The stochastic hill climbing algorithm used for creating strings of weight vectors for the MO-PTSP is relatively simple, although it was able to obtain very good results in few iterations. However, it could be desirable to model behaviours that dynamically change the weight vectors according to measurements of the game state, instead of evolving a different solution for each particular map. A possible extension for this work is to analyse and discover features in the game state that allows the establishment of relationships between game situations and the weight vectors to use for each objective. This mechanism would be general enough to be applicable to many different maps without requiring specific learning in any particular level.

Finally, the results described in this paper allow us to assume that multi-objective approaches can provide a high quality level of play in real-time games. Multi-objective problems pop up in different settings and by using MCTS we can balance not only between exploration and exploitation but between multiple objectives as well.

Acknowledgments

This work was supported by EPSRC grants EP/H048588/1, under the project entitled "UCT for Games and Beyond". The authors would also like to thank Edward Powley and Daniel Whitehouse for their work in the controller `PurofMovio` for the MO-PTSP competition.

References

[1] R. T. Marler and J. S. Arora, "Survey of Multi-objective Optimization Methods for Engineering," *Structural and Multidisciplinary Optimization*, vol. 26, pp. 369–395, 2004.

[2] C. Coello, *Handbook of Research on Nature Inspired Computing for Economy and Management*. Idea Group Publishing, 2006, ch. Evolutionary Multi-Objective Optimization and its Use in Finance.

[3] D. Perez, S. Samothrakis, and S. Lucas, "Online and Offline Learning in Multi-Objective Monte Carlo Tree Search," in *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, 2013, pp. 121–128.

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

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

[6] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy, "Progressive Strategies for Monte-Carlo Tree Search," *New Math. Nat. Comput.*, vol. 4, no. 3, pp. 343–357, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.9239&rep=rep1&type=pdf

[7] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," *Machine Learning: ECML 2006*, vol. 4212, pp. 282–293, 2006.

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

[9] 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*, Milan, Italy, 2009, pp. 249–255.

[10] 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*, vol. 3, no. 2, pp. 142–154, 2011.

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

[12] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. M. J.-B. Chaslot, and J. W. H. M. Uiterwijk, "Single-Player Monte-Carlo Tree Search," in *Proceedings of Computer Games, LNCS 5131*, 2008, pp. 1–12.

[13] T. Cazenave, "Reflexive Monte-Carlo Search," in *Proc. Comput. Games Workshop*, Amsterdam, Netherlands, 2007, pp. 165–173.

[14] 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*, pp. 1–16, 2013.

[15] E. J. Powley, D. Whitehouse, and P. I. Cowling, "Monte Carlo Tree Search with Macro-Actions and Heuristic Route Planning for the Physical Travelling Salesman Problem," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 234–241.

[16] ——, "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.

[17] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

[18] C. Coello, "An Updated Survey of Evolutionary Multiobjective Optimization Techniques: State of the Art and Future Trends," in *Proc. of the Congress on Evolutionary Computation*, 1999, pp. 3–13.

[19] A. Zhou, B.-Y. Qu, H. Li, S.-Z. Zhao, P. N. Suganthan, and Q. Zhang, "Multiobjective Evolutionary Algorithms: A Survey of the State of the Art," *Swarm and Evolutionary Computation*, vol. 1, pp. 32–49, 2011.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2002.

[21] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. A Bradford Book, 1998.

[22] P. Vamplew, R. Dazeley, A. Berry, R. Issabekov, and E. Dekker, "Empirical Evaluation Methods for Multiobjective Reinforcement Learning Algorithms," *Machine Learning*, vol. 84, pp. 51–80, 2010.

[23] Z. Gabor, Z. Kalmar, and C. Szepesvari, "Multi-criteria Reinforcement Learning," in *The fifteenth international conference on machine learning*, 1998, pp. 197–205.

[24] S. Natarajan and P. Tadepalli, "Dynamic Preferences in Multi-Criteria Reinforcement Learning," in *In Proceedings of International Conference of Machine Learning*, 2005, pp. 601–608.

[25] L. Barrett and S. Narayanan, "Learning All Optimal Policies with Multiple Criteria," in *Proceedings of the international conference on machine learning*, 2008, pp. 41–47.

[26] E. Zitzler, *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. TIK-Schriftenreihe Nr. 30, Diss ETH No. 13398, Swiss Federal Institute of Technology (ETH) Zurich: Shaker Verlag, Germany, 1999.

[27] W. Weijia and M. Sebag, "Multi-objective Monte Carlo Tree Search," in *Proceedings of the Asian Conference on Machine Learning*, 2012, pp. 507–522.

[28] ——, "Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search," *Machine Learning*, vol. 92:2–3, pp. 403–429, 2013.

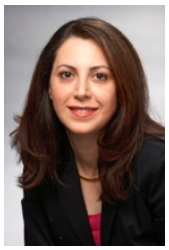
[29] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and Move Groups in Monte Carlo Tree Search," in *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2008, pp. 389–395.

[30] T. Kozelek, "Methods of MCTS and the game Arimaa," M.S. thesis, Charles Univ., Prague, 2009.

[31] 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, pp. 1–8.

[32] ——, "Monte Carlo Tree Search: Long Term versus Short Term Planning," in *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 2012, pp. 219 – 226.

Diego Perez is currently pursuing a Ph.D. in Artificial Intelligence applied to games at the University of Essex, Colchester. He has published in the domain of Game AI, participated in several Game AI competitions and organized the Physical Travelling Salesman Problem competitions, held in IEEE conferences. He also has programming experience in the videogames industry with titles published for game consoles and PC.



Sanaz Mostaghim is a professor of computer science at the Otto von Guericke University Magdeburg, Germany. Her current research interests are in the field of evolutionary computation, particularly in the areas of evolutionary multi-objective optimization, swarm intelligence and their applications in science and industry. She is an active IEEE member and is serving as an associate editor for IEEE Transactions on Evolutionary Computation and IEEE Transactions on Cybernetics.

Spyridon Samothrakis is currently pursuing a PhD in Computational Intelligence and Games at the University of Essex. His interests include game theory, computational neuroscience, evolutionary algorithms and consciousness.

Simon Lucas (SMIEEE) is a professor of Computer Science at the University of Essex (UK) where he leads the Game Intelligence Group. His main research interests are games, evolutionary computation, and machine learning, and he has published widely in these fields with over 160 peer-reviewed papers. He is the inventor of the scanning n-tuple classifier, and is the founding Editor-in-Chief of the IEEE Transactions on Computational Intelligence and AI in Games.