Online and Offline Learning in Multi-Objective Monte Carlo Tree Search

Diego Perez, Spyridon Samothrakis and Simon Lucas
School of Computer Science and Electronic Engineering
University of Essex, Colchester CO4 3SQ, UK
dperez@essex.ac.uk, ssamot@essex.ac.uk, sml@essex.ac.uk

***Abstract*—Multi-Objective optimization has traditionally been applied to manufacturing, engineering or finance, with little impact in games research. However, its application to this field of study may provide interesting results, especially for games that are complex or long enough that long-term planning is not trivial and/or a good level of play depends on balancing several strategies within the game.**

This paper proposes a new Multi-Objective algorithm based on Monte Carlo Tree Search (MCTS). The algorithm is tested in two different scenarios and its learning capabilities are measured in an online and offline fashion. Additionally, it is compared with a state of the art multi-objective evolutionary algorithm (NSGA-II) and with a previously published Multi-Objective MCTS algorithm. The results show that our proposed algorithm provides similar or better results than other techniques.

I. Introduction

Viewed simplistically, most competitive games can be thought of as having a single objective: to win. While this is easily stated, achieving it is usually complex, otherwise the game would be of limited interest. Successful approaches often include using some form of heuristic tree search, where the heuristic may be defined as a weighted sum of features of the game state. For example, a simple chess heuristic attaches different weights to each piece in line with their value.

However, multi-objective optimization has been a field of study mainly within enginering [9] and finance [4] applications, and these algorithms have had little impact on games research. We argue that multi-objective approaches have much to offer in developing richer game strategies with more fine-grained control of alternative policies.

In real-time strategy games (for instance, *Starcraft*) where the objective is clearly defined (being the only one alive at the end of the match), the workarounds to achieve the victory may take several factors into account simultaneously, such as balancing attack power, defensive structures and resource gathering.

When building up a team in a role-playing game (RPG), the different members need to be balanced to be competitive (strength, dexterity, and healing capabilities, for instance). Multi-objective approaches may help in these scenarios, where different goals have to be attended to at the same time.

Furthermore, even simpler games can benefit from these approaches. For instance, in *Othello*, a player might want, in a mid-game scenario, to maximize mobility (i.e. number of available moves to make) as well as trying to play in the corners, as this has proven to be a good strategy.

This paper presents a new algorithm that combines Multi-Objective techniques with Monte Carlo Tree Search (MCTS), a Reinforcement Learning (RL) algorithm that has become more and more popular within the last decade. The proposed algorithm is tested in two different simple games, *Deep Sea Treasure*, and *Puddle Driver*, and its performance is compared with state of the art multi-objective approaches. Additionally, the algorithm is analyzed in two different ways: in an offline (approximating solutions from an initial state) and online (actually playing the game) learning modes.

The paper is structured as follows. First, Section II provides the necessary background for Multi-Objective optimization techniques. Section III describes MCTS and the only, to the best of our knowledge, multi-objective MCTS implementation in the literature. In Section IV, the algorithm proposed in this paper is introduced, followed by the description of the two games employed in this research, in Section V. Then, the experimental setup and analysis of results are detailed in Section VI. Finally, conclusions and future work are discussed in Section VII.

II. Multi-objective optimization

A multi-objective optimization problem (MOOP) is presented in a scenario where two or more objective functions are to be optimized. In its general form a MOOP is described as the minimization (or maximization) of a function $F_m(x)$, where $x = (x_1, x_2, \ldots, x_n)$ is an element of the *decision space* and $F_m(x) = (f_1(x), f_2(x), \ldots, f_m(x))$ belongs to the *solution space*. In other words, each solution to the problem is a vector of n variables that provides m different scores (or rewards, or fitness) that are meant to be optimized.

A solution $F_m(x)$ is said to *dominate* another solution $F_m(y)$ if it is not worse than $F_m(y)$ in all objectives for all $i = 1, 2, \ldots, m$, and at least one objective of $F_m(x)$ is better than its analogous counterpart in $F_m(y)$. If these conditions apply, it is said that $F_m(x) \preceq F_m(y)$ ($F_m(x)$ dominates $F_m(y)$, and $F_m(x)$ is non-dominated by $F_m(y)$). This definition allows a comparison between two solutions from the solution space: if $F_m(x) \preceq F_m(y)$, then $F_m(x)$ is considered to be better than $F_m(y)$.

In those cases where it cannot be said that $F_m(x) \preceq F_m(y)$ (when, for instance, $F_1(x) < F_1(y)$ but $F_2(x) > F_2(y)$), these solutions are non-dominated with respect to each other. The set of the solutions that are non-dominated is called the *non-dominated set*. Given a non-dominated set P, it is said that P

978-1-4673-5311-3/13/$31.00

is the *optimal pareto front* if there is no other solution in the solution space that dominates any member of P.

An important question that arises is how to measure the quality of a given pareto front. A possibility is to use the Hypervolume Indicator (HV): 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 $\mathbb{R}^d$. For instance, if the objectives are to be maximized, the higher the $HV(P)$, the better the front calculated.

Multiple algorithms have been proposed in the literature to tackle multi-objective optimization problems. One of the simplest and most used methods is the weighted-sum approach: each objective is given a weight by the user and the problem is scaled to a single-objective optimization. By providing different values for the weights, it is possible to converge to any of the solutions of the optimal pareto front if this is convex. However, as described by Deb [6], linear scalarisation approaches fail to find solutions in the non-convex case. For more extensive descriptions, definitions, properties and multi-objective optimization in general, the reader can consult the work by Deb [6].

A. Evolutionary Approaches to MOOP

Evolutionary multi-objective optimization (EMOA) algorithms have become a popular choice to approach multi-objective optimization problems [3], [16]. One of the most mentioned algorithms in the literature is the Non-dominated Sorting Evolutionary Algorithm 2 (NSGA-2, also employed in this research). NSGA-2 presents a genetic algorithm that evolves a population of individuals, which is ranked according to dominance criteria and crowding distance.

NSGA-2 is based on three main concepts: a *fast non-dominated sorting* algorithm to rank the individuals of the population and group them in pareto fronts; each individual is assigned a *crowding distance* value, that measures how close it is to its neighbours. This value is used, along with its rank in the population, to apply the selection genetic operator. Finally, the algorithm implements *elitism*, by automatically promoting the best N individuals to the next generation. A full description of the algorithm can be found in [5].

B. Reinforcement Learning Approaches to MOOP

Multi-objective optimization has also been a matter of study for Reinforcement Learning (RL) algorithms. RL [13] is a broad field in Machine Learning that copes with situations where an agent has to discover which actions (or sequences of actions) should be applied in order to maximize the reward.

An RL problem can be defined with the tuple[1] (S, A, T, R, π). S represents the set of states, where s_0 is the initial state. A is the set of available actions the agent can make, and the transition model $T(s_i, a_i, s_{i+1})$ determines the probability of reaching the state s_{i+1} after applying action a_i in s_i. The reward function $R(s_i)$ provides a single number (*reward*) that the agent must optimize, and it represents the desirability of the state s_i reached. Finally, a decision policy $\pi(s_i) = a_i$ maps states to actions, determining which actions must be chosen from each state $s \in S$. One of the most important challenges in RL is the trade-off between exploration and exploitation. The decision policy can choose between following actions that provided good rewards in the past versus exploring new parts of the search space by selecting new actions.

Multi-objective Reinforcement Learning (MORL) [14] modifies this definition by using a vector $R = r_0, r_1, \ldots, r_n$ for the rewards of the problem. In other words, MORL problems differ from RL in having $n > 1$ objectives that must be optimized. If the different objectives are independent or they are not in conflict, approaches like the scalarization technique described above would be suitable to solve the problem by using a conventional RL algorithm on a single objective obtained by a weighted-sum of the multiple rewards. However, this is not always the case, as it is usual that the objectives are in conflict and the policy (π) must balance among them.

III. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search technique that builds a tree in memory with the outcomes of Monte Carlo (MC) simulations. MCTS is usually employed as an online planning technique: the plan (actions to perform) is recalculated every time step. Given an instant t, several iterations of the algorithm are performed to plan up to a certain point in the future. Then, an action is chosen, executed and the next moved is re-planned from one step forward.

The best known version of MCTS is the Upper Confidence Bounds for Trees (UCT), firstly introduced by Kocsis and Szepesvári [7], that employs UCB1 (see Equation 1) as a tree selection policy. This policy balances the exploitation of known moves (first term of Equation 1) and the exploration of a new portion of the search space (second term). $Q(s, a)$ is the empirical average of the rewards obtained by choosing action a in state s. $N(s, a)$ represents the number of times the action a was chosen from state s, and $N(s)$ counts how many times state s was visited in the tree. Finally, C is a constant that balances both terms. The value for this constant depends on the problem approached, but it is common to find $C = \sqrt{2}$ in single-player games (or puzzles) and $C \simeq 1$ in two-player games, assuming rewards are normalized in the range $[0, 1]$.

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

The algorithm works as follows: starting from the root node (current state), the agent applies the tree policy (*selection* phase) to navigate to a node in the tree which is not fully expanded. Once at a leaf node, the algorithm creates a new node (*expansion*) and applies an MC simulation until reaching an end-game state (*simulation* step). Finally, the reward r obtained is back-propagated through the visited nodes until reaching the root (*Backpropagation*). On each node, all the statistics ($Q(s, a)$, $N(s, a)$ and $N(s)$) are updated for the next selection step. This process is repeated until a termination criteria is met (such as number of iterations or elapsed time).

[1] The RL problem can be formalised further using Markov Decision Processes (or Partially Observable Markov Decision Processes), but we will not use these definitions here.

Finally, the algorithm chooses which is the next action to make based on the statistics of the root node.

The success of MCTS in recent years is due to it being the first algorithm able to provide a professional level of play in the game of Go for small board sizes (9×9). Since that achievement, MCTS has been widely used in many different games and applications. C. Browne et al. [1] presented a recent survey where the algorithm, its applications and variants are described.

A. Multi-objective Monte Carlo Tree Search

Moving from single to multi-objective MCTS (MO-MCTS) requires some modifications in the baseline algorithm. The most immediate modification is to the rewards obtained at the end of an MC simulation: instead of retrieving a single value r, the new reward is a vector $R = r_1, r_2, \ldots, r_n$. Therefore, the value $Q(s,a)$ becomes a vector that keeps the empirical average of n rewards. The meaning of the other statistics ($N(s,a)$ and $N(s)$), however, do not change, as they just indicate how many times each node is traversed. The question to answer then is how to use $Q(s,a)$ in Equation 1, given that it is now a vector instead of a scalar value.

The first (and, to the best of our knowledge, only) application of MCTS to a multi-objective domain has been performed by W. Wang and M. Sebag [15]. In their work, the authors replace the UCB1 equation from UCT with a mechanism based on the HV calculation. During the execution of the algorithm, a Pareto Archive (P) front is kept with all solutions found in terminal states for the given scenario. In every node in the tree, $\overline{r}_{sa}$ is defined as a vector of UCB1 values, in which each element is the result of calculating UCB1, where $Q(s,a)$ is defined for each objective (i.e. each $\overline{r}_{sa,i}$ is calculated using the rewards for objective i). The next step is to define $W(s,a)$ as in Equation 2:

$$W(s,a) = \begin{cases} HV(P \cup \overline{r}_{sa}) - dist(\overline{r}^{p}_{sa}, \overline{r}_{sa}) & Otherwise \\ HV(P \cup \overline{r}_{sa}) & if \;\; \overline{r}_{sa} \preceq P \end{cases} \tag{2}$$

where $\overline{r}^{p}_{sa}$ is the projection of $\overline{r}_{sa}$ into the piecewise linear surface defined by the pareto archive P. 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 $\overline{r}^{p}_{sa}$ is subtracted from the HV calculation. The tree policy selects actions based on a maximization of the value of $W(s,a)$.

The proposed algorithm implemented two known heuristics: Rapid Action Value Estimate (RAVE) and Progressive Widening (PW) (see [1], [15] for a description of these). MO-MCTS was employed successfully in two domains: the DST and the Grid Scheduling problem, matching state of the art results in both domains, at the expense of a high computational cost.

IV. Pareto MO-MCTS

This section describes the algorithm proposed in this paper. As mentioned in Section III, the vector of rewards $\overline{r}$ obtained at the end of an MC simulation is propagated through the visited nodes until reaching the root. Each one of the nodes will update the accumulated reward in a vector $\overline{R}$, as in the vanilla MCTS algorithm. However, in the algorithm proposed here, each node will also keep a local pareto front (P) that is updated with every vector of rewards $\overline{r}$ obtained during the simulations. If $\overline{r}$ is not dominated by the local pareto front, it is added to it and propagated to its parent. If $\overline{r}$ is dominated by the existing pareto front in one of the nodes, it is not added to it. Algorithm 1 provides an example of this procedure.

Algorithm 1 Pareto MO-MCTS node update.

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

This mechanism has several implications: first of all, 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). Secondly, by construction, if a reward $\overline{r}$ is dominated by the 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 cost in the efficiency of the algorithm. Finally, it is easy to infer, from the last point, that the 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 reasons. First, the algorithm can store information in the root that indicates which is the best action to take in order to converge to different points of the front discovered. As described in Section III, MCTS can be used in an online manner: the algorithm performs its search for a given number of iterations and then a move needs to be chosen. Hence, in this implementation, the root stores information about which points in its non dominated front are reached by following its children. This data is used in the experiments shown in Section VI-B. Secondly, the pareto front of the root can be used to measure the quality of the search, using, for instance, the hypervolume calculation.

However, there is still one issue that needs to be defined: how the tree policy uses the value of $Q(s,a)$. Based on the concept of HV, two different values for $Q(s,a)$ are explored in this research: the first one, **HV-MO-MCTS**, defines $Q(s,a) = HV(\overline{R})/N(s)$. The second option, called here **Pareto-MO-MCTS**, employs $Q(s,a) = HV(P)/N(s)$. While the first one uses the HV of the averaged reward, the second case employs the HV of the local pareto front P of the node.

There are several advantages of the approach introduced here over the algorithm presented in III-A. First, it is computationally much less expensive as there is no need to calculate the piecewise linear surface of the pareto front. This is particularly important for real-time games with a reduced time budget to choose a move. Secondly, the algorithm presented in this paper

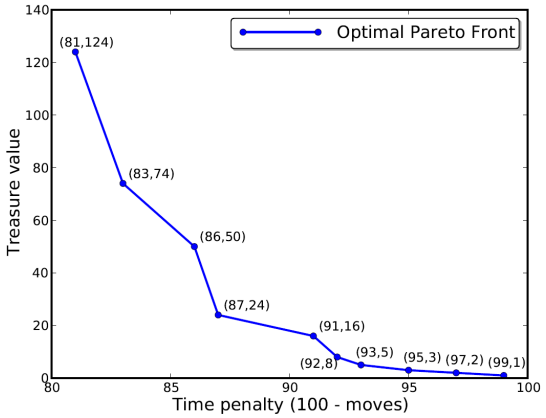

Fig. 1: Optimal Pareto Front of the Deep Sea Treasure, with both objectives to be maximized. From [15].

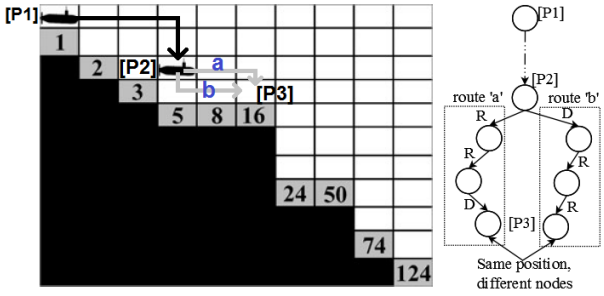

Fig. 2: Example of two different sequences of actions (R: Right, D: Down) that lie in the same position in the map, but different node in the tree.

does not include any heuristic in its vanilla form (such as RAVE or PW), making it simpler and more applicable to a wider range of domains. Finally, the root of the tree contains the best pareto front found during the search, and it can store information to indicate which action leads to what solution in the discovered front.

V. Benchmarks

A. Deep Sea Treasure

The Deep Sea Treasure (DST) is a multi-objective problem introduced by Vamplew et al. [14]. The DST is an episodic single-player puzzle where a submarine must find a treasure at the bottom of the sea by moving in a grid of 11 rows and 10 columns. The vessel can perform four different moves (*up*, *down*, *right* and *left*). In case the action applied takes the ship off the grid or into the sea floor, the submarine's position does not change.

Figure 2 (on the left) shows the environment of the DST. The vessel starts at the top left corner of the grid and the 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.

This puzzle has two objectives: to minimize the number of moves performed while maximizing the value of the treasure found. Furthermore, the submarine can make no more than a maximum of 100 moves, which allows the problem to be defined as the maximization of two scores: $(100 - moves$, *treasure value*). The reward obtained for each location where there is no treasure is $(-1, 0)$.

Figure 1 shows the optimal pareto front of the problem. This front contains 10 non dominated solutions, one for each treasure in the puzzle. As can be seen, the pareto front of the DST is globally concave, containing 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.

DST is an interesting problem for multi-objective optimization because of the concave shape of its optimal pareto front. As mentioned earlier in Section II, this is a hazard for linear scalarisation approaches. Concretely, in the DST, such approximations converge to the non dominated solutions at the edges of the pareto front: $(81, 124)$ and $(99, 1)$. Thus, successful approaches should be able to find all elements of the optimal pareto front and potentially converge to any of the non dominated solutions.

1) Transposition Tables in DST: One of the features of the DST is that it allows the usage of Transposition Tables (TT) [2]. TT is a technique used to optimize the search in tree search based algorithms. It is based on the fact that the same state can be found in different locations within the tree. Figure 2 shows an example of this situation in the DST.

Let us imagine that the submarine starts in position $P1$ (root of the tree) and moves right and down until reaching position $P2$. From this point, in order to reach $P3$, the vessel could take two different routes (in this example, whereas in the general case, there are many more possibilities): route a (applying actions *Right*, *Right*, *Down*) or route b (*Down*, *Right*, *Right*). Both routes end in the same position, $P3$, but two different nodes represent that position in the tree. It is important to notice that the coordinates of the positions alone are not enough to identify equivalent situations. Imagine a third route c that, from $P2$, executes the following moves: *Up*, *Right*, *Right*, *Down*, *Down*. The final location would also be $P3$, but in this case the submarine would have performed 5 moves, instead of the 3 moves that routes a and b made. From the point of view of the tree, the node where $P3$ was found would be two levels deeper in the tree.

It is sensible to presume that sharing information between equivalent situations should help the search procedure, as otherwise these nodes would be treated as two distinct positions. In the algorithm presented in this paper, TT are implemented with a hash map that stores a *representative* node for those equivalent positions. The key of the hash map needs to be calculated using three values: coordinates x and y of the position and depth in the tree. This way, transpositions can only occur at the same depth of the tree. This feature has been previously explored in the literature with success [8].

The usage of TT modifies the proposed algorithm for the DST. In this case, every time a new node is added to the tree (*expansion* phase), the algorithm checks first if an equivalent node is already stored in the TT. If not, the node is created

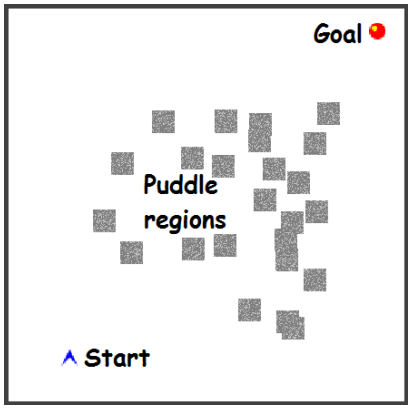

Fig. 3: Example map from Puddle Driver.

and a new entry is added to the table. If there is already a node in the table, the parent node points to the existing node without the need of creating a new one. Additionally, the *Back-propagation* step needs to be modified in order to update the actual nodes traversed from the route, using a list of nodes instead of just following the parents.

B. Puddle Driver

The second benchmark used in this paper is the Puddle Driver. This problem, created explicitly for this research, is a single-player real-time puzzle that has influence from the Multi-Objective PuddleWorld (MOP) and the Physical Travelling Salesman Problem (PTSP). The Multi-Objective PuddleWorld, presented also by Vamplew et al. [14], offers a puzzle where an agent has to reach a goal in a grid based world where puddles lie between its starting position and the destination. The target is to minimize two objectives: the number of movements to reaching the goal and the presence of the agent in the puddles. In the PTSP, a real-time game fully described in [11], the player must drive a ship in a continuous space in order to visit a series of waypoints scattered around the world.

The Puddle Driver presents a real-time scenario where the player drives a ship by supplying 6 different actions, like in the PTSP, combining two different inputs: *thrust* (with values *on* and *off*) and *steer* (rotating *left*, *right* or *straight*). The ship must apply an action every $40ms$, and this move will alter its position, velocity and orientation. The initial position of the ship is located at the bottom left corner of the map, while a target waypoint is placed at the top right corner. The size of the map is 512×512 pixels. There are several regions (puddles) randomly placed between these two positions, and they cause 1 unit of damage for each time step the ship drives over them. Analogously to MOP, the objective is to minimize both quantities: time taken to reach the goal, and damage suffered in the puddles. The game ends if, after 1000 time steps, the goal is not reached. Figure 3 shows an example of a map for this game.

The Puddle Driver has several features that make it interesting and more challenging than the DST. First of all, it takes place in a continuous world, so discretizations obtained with techniques such as transposition tables used here for the DST (see Section V-A1) are not directly applicable to this game. Secondly, it usually takes more than 300 steps to reach the goal. Considering the $40ms$ of time allowed per game step, creating plans for the whole route becomes complex. Therefore, it is necessary to be able to measure the quality of a given position even, if in the move plan in a determined game step, the goal is not yet reachable.

The reward scheme provided for the Puddle Driver consists of multiplying each one of the two objectives (*total time* and *damage*) by the distance from the current position of the ship to the goal's location. As, given the shape and size of the map, the maximum distance between two points is the length of the diagonal ($MD \simeq 725$), a factor f in the range [0,1] is obtained by calculating $f = 1 - \frac{dist}{MD}$, where $dist$ is the current distance between ship and goal. Thus, the reward vector $\overline{r}$ at time step t can be defined as $\overline{r_t} = \{(1000 - T) \times f, (1000 - D) \times f\}$, where T (*total time*) and D (*damage*) are the two rewards of the problem, and $\overline{r_t}$ is meant to be maximized.

1) Macro-actions for Puddle Driver: Previous research in PTSP [12], [10] shows that the use of macro-actions for real-time navigation domains increases the performance of the algorithms. Given the similarities (identical actions and generative models) between PTSP and the Puddle Driver, it is straightforward to assume that macro-actions will be equally effective in the benchmark presented here. There are several possibilities when using macro-actions, but in this research the simplest approach has been taken: a macro-action of length L is defined as a sequence of L repetitions of the same action.

Plans composed of macro-actions can see further forward in time than those composed only of single actions. This reduces the search space drastically and allows for a better algorithm performance. The length of macro-action for this research is $L = 15$, a value that has previously shown its proficiency 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 [10].

VI. Experimentation

Two different approaches have been taken for the experiments presented in this paper: a) *offline* learning, where the focus is on how the optimal pareto front is approximated; and b) *online* learning, with the focus set on how the algorithms are able to play the game.

A. Offline learning

The performance of the algorithms tested is compared, showing how fast are they able to find the optimal pareto front. In order to do this, the HV of the pareto front found by the algorithms is shown in relation to the training iterations performed (that is, simulations in MCTS approaches and evaluations in NSGA-II).

Four different algorithms are compared: Pareto-MO-MCTS, HV-MO-MCTS (both, as described in Section IV, with TT) and NSGA-II.[2] For the latter one, two versions with 50 and 100 individuals per population have been tried. Each individual's genome is a sequence of $N = 100$ actions to

[2]NSGA-II from library "MOEA Framework" - www.moeaframework.org

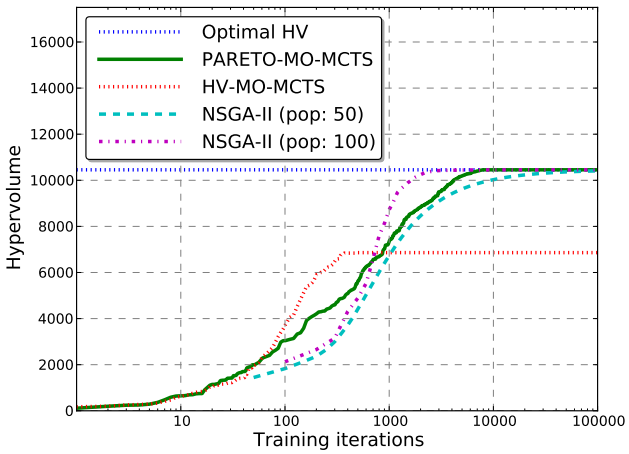

Fig. 4: Offline training. Hypervolume of the solution versus training iterations in logarithmic scale. The optimal hypervolume indicator for the DST is 10455, firstly reached by Pareto-MO-MCTS. NSGA-II (pop: 100) achieves sooner a very close hypervolume of 10400, although it does not converge to the optimal HV until reaching close to 19000 iterations.

Iterations:	10	100	1000	10000
Pareto-MO-MCTS (HV $\sim$)	750	3000	7000	10455
MO-MCTS [15] (HV $\sim$)	1000	2000	4250	9500

TABLE I: HV in Pareto-MO-MCTS and MO-MCTS [15].

make (as this is the maximum number of moves), represented by integers, a value per action. All algorithms are run 40 times and the average results are shown in Figure 4.

One initial result to note is that all algorithms but one are able to converge to the optimal, NSGA-II (in both versions) and Pareto-MO-MCTS. The first to achieve the optimal pareto front is Pareto-MO-MCTS, in $\sim$ 8000 training steps. NSGA-II with 100 population size performs very well, getting to a hypervolume indicator of 10400 in around $\sim$ 2500 iterations. However, it does not get to the optimum until reaching $\sim$ 19000 steps. If the population size is reduced to 50, NSGA-II is not able to converge to the optimum before 100000 iterations.

It is worthwhile highlighting that HV-MO-MCTS is the algorithm whose HV indicator grows the fastest, until a maximum ($HV \sim 6800$), which is quite far from the optimal. Both MO-MCTS algorithms produce higher HV indicators than the NSGA-II ones in early stages of the runs, although only Pareto-MO-MCTS is able to find the optimal pareto front.

Another interesting comparison is the performance obtained with the Pareto-MO-MCTS algorithm presented here and the one from W. Wang and M. Sebag [15], described in section III-A. The approach presented in their paper converges to the optimal HV indicator beyond 100000 iterations (averaged over 11 independent runs). Pareto-MO-MCTS seems to converge faster as well, Table I shows an approximation of the HV indicators obtained in these approaches.

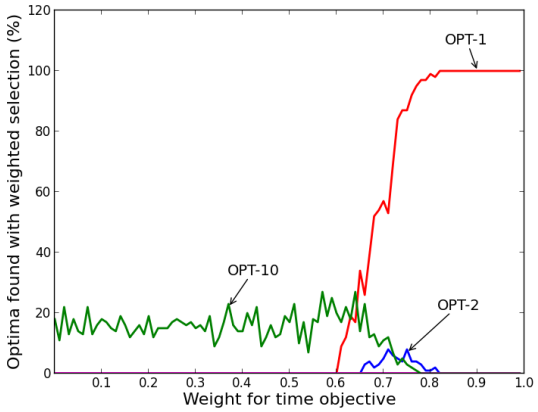

Fig. 5: Linear scalarization for move selection. Percentage of optimal points found for values of weights in [0,1].

B. Online learning: DST

In this and the following sections, the algorithms are tested in a different way: each approach has $40ms$ to spend in a training phase before deciding a move to make. When the submarine has moved, the process repeats, re-planning again the next move within another $40ms$ (The experiments were run in an Intel Core i5, 230GHz, 4GB of RAM). For NSGA-II (resp. Pareto-MO-MCTS), the population (resp. the tree) is reset at every game cycle.

Both algorithms, NSGA-II and Pareto-MO-MCTS, provide the following information at the time an action needs to be chosen: a non-dominated front; and which actions (first gene in a NSGA-II individual, a root's child in MCTS) lead to what points in the front. The usage of evolutionary algorithms for online learning in games is not new, and it has shown promising results [12].

The question is how actions must be chosen to make the next move. A straightforward idea could be to use values to weight both objectives, and then choose the action that leads to the point in the discovered pareto front that maximizes the weighted sum. However, as explained earlier in Section V-A, the optimal pareto front of the problem is non convex, so a linear scalarization approach would not find all solutions in the pareto front (even if they were all discovered by the algorithm!). Figure 5 shows an example of this phenomenon, with Pareto-MO-MCTS executed in an online fashion with $40ms$ per move (similar results are obtained with NSGA-II). 100 games are executed for each value of the weights that range from 0 to 1 in increments of 0.01. There are clearly two points where the submarine ends the game, OPT-1 (99,1) and OPT-10 (81, 124), which are the edges of the optimal pareto front of the DST, as foreseen in Section V-A. Rewards are normalized in the range (0,1) to perform the weighted sum.

In order to overcome this problem, the following move selection algorithm has been implemented: using the non dominated front provided by the algorithm, it is possible to define a pair of weights $V = (v_1, v_2)$, linear-normalized in the range $(0, 1)$, and calculate the distance from V to each point in the pareto front. The action to choose would be the

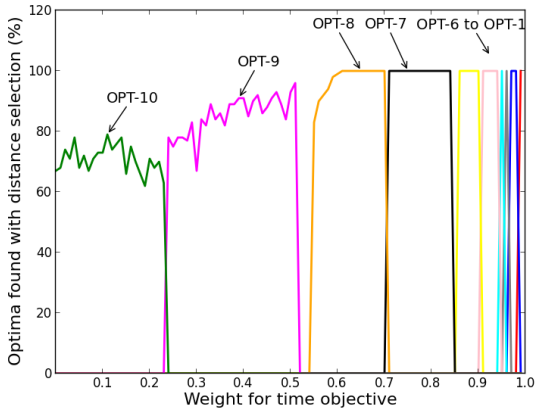

Fig. 6: Distance based move selection in Pareto-MO-MCTS: Percentage of optima points found for weights in [0,1].

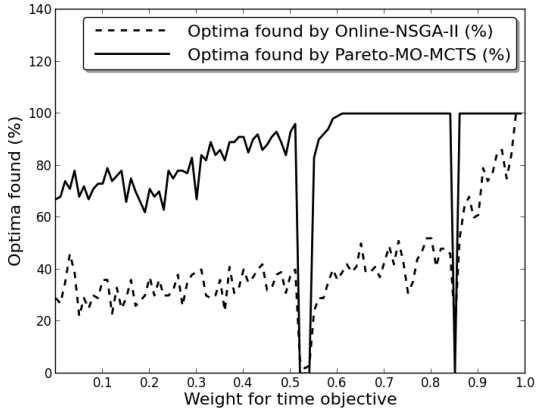

Fig. 7: Percentage of optima found in DST against weights.

one that leads to the point in the pareto front with the smaller Euclidean distance to V. The results obtained by applying this action selection mechanism to Pareto-MO-MCTS are depicted in Figure 6. As can be seen, all points in the optimal pareto front are found in the range of the values of V chosen.

The optimal pareto fronts 1 to 8 are obtained with a very high rate, while the optima 9 and 10 are achieved 90% and 70% of the time with the appropriate weights. The explanation for this is that optima points 9 and 10 are the ones furthest from the starting location of the submarine, and hence more moves are needed to find the optima. Another interesting feature is the fact that not all objectives are found with the same number of possible weights, which is a consequence of the shape of the optimal front.

Figure 7 shows a comparison between Pareto-MO-MCTS and NSGA-II. To make this picture clearer, the image represents the optima found without making distinctions as to which optima the vessel ends in. Instead, the picture shows the accumulated percentage of all optima achieved for each weight value.

With the $40ms$ of allocated time for the training phase, Pareto-MO-MCTS clearly outperforms NSGA-II, driving the submarine to points in the pareto optima front more often. This result is due to the better performance of Pareto-MO-MCTS versus NSGA-II. In $40ms$, Pareto-MO-MCTS is able to execute ~ 4500 iterations, while NSGA-II performs ~ 1250 evaluations.

C. Online learning: Puddle Driver

As described earlier, the search space of the Puddle Driver is so vast that it is not possible to plan the full trajectory to follow in real-time (i.e. offline learning is infeasible). Hence, this problem has been used as a secondary (and harder) test for the algorithm presented in this paper. The optimal pareto front for this problem is unknown, so the focus in these experiments is on how different weights provide different solutions, and the comparison between the results obtained by each one of the algorithms presented here.

In order to evaluate the performance of these algorithms in the Puddle Driver, 10 different maps are created with 25 puddle squares, of 25×25 pixels, uniformly randomly distributed in them. Each algorithm is executed 10 times on each map, providing a total of 100 executions for each one of the weights employed. For this problem, the weights range from 0 to 1 in increments of 0.1.

The population size for NSGA-II has been set to 50, a value determined empirically. Both algorithms execute $N = 8$ macro-actions per evaluation (i.e. in NSGA-II, the length of the individual's genome is 8), where the macro-action length is $L = 15$, leading to $8 \times 15 = 120$ single actions applied to evaluate the individual.

Figure 8 shows the average rewards (with standard error) obtained by NSGA-II and Pareto-MO-MCTS for each one of the values tested for the weights. As can be seen, the total time spent (upper sub-figure from Figure 8) decreases as long as this objective gets a higher reward, while the contrary happens with the damage taken in puddle regions (lower plot). In most cases, Pareto-MO-MCTS obtains better rewards than NSGA-II.

Figure 9 shows the averages of the solutions found in a two-dimensional view. It can be observed that the Pareto-MCTS algorithm obtains better solutions than the online version of NSGA-II, providing a more diverse range of solutions weighting between both objectives.

VII. Conclusions

This paper presented a new Multi-Objective MCTS algorithm for games. It has been tested in two different domains (*Deep Sea Treasure* and *Puddle Driver*), using two different approaches (*online* and *offline* learning), and compared against a state of the art multi-objective algorithms (NSGA-II) and a previously published algorithm for MO-MCTS. The latter one is only tested in the offline scenario, as this approach is not suited for the real-time constraints present in the online cases, because of its high computational cost.

The algorithm presented here shows its proficiency in all the scenarios tested. For the *offline* case, tested on the DST, it provides comparable performance to one of the NSGA-II versions and improves the results obtained by previous research in

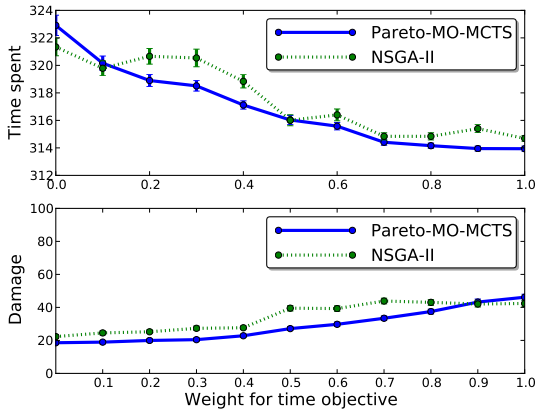

Fig. 8: Average rewards with standard error against weights for first objective. Smaller values are better.

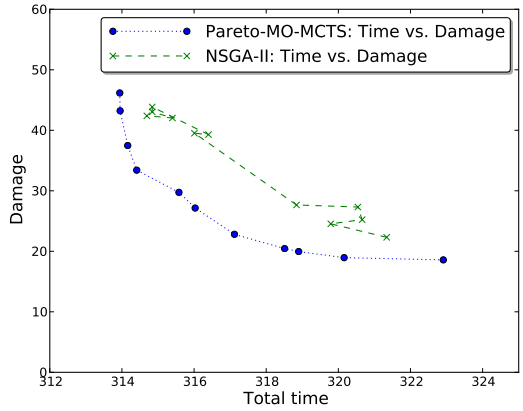

Fig. 9: Average solutions found for the Puddle Driver.

to MO-MCTS by providing a faster learning rate. In the *online* scenario, the algorithm presented here obtains better results than NSGA-II, showing better planning capabilities when the time given to make a move is reduced. Two different versions of the same algorithm are analyzed in this paper. *Pareto-MO-MCTS* produces better results in the *offline* scenario than *HV-MO-MCTS*, although both of them have shown a similar performance in the *online* case.

Immediate future work to continue this research would be to explore a more complex and challenging scenario. For instance, the Multi-Objective Physical Travelling Problem [3], a game that features in a competition in the 2013 IEEE Conference on Computational Intelligence and Games (CIG), would be a perfect choice for this. It is also possible that the differences between *Pareto-MO-MCTS* and *HV-MO-MCTS* become bigger when the benchmark used is more complex.

The algorithm itself is also a matter for future work. For instance, the gaps present in Figures 6 and 7 suggest that some optima points in the DST are not accessible if determined weights are given. Maybe an alternative procedure for choosing the action to make at each game step could avoid this phenomenon. Another possibility is to incorporate the concept of *crowding distance* into the algorithm, as algorithms such as NSGA-II do, in order to provide a better distribution of solutions on the Pareto front of each node.

[3]www.ptsp-game.net

ACKNOWLEDGMENT

This work was supported by EPSRC grant EP/H048588/1. The authors also thank Weijia Wang and Michèle Sebag for sharing their research, code and experiments with us.

REFERENCES

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

[2] Benjamin E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and Move Groups in Monte Carlo Tree Search. In *Proceedings of IEEE Symposium on Computational Intelligence and Games*, pages 389–395, 2008.

[3] Carlos Coello. An Updated Survey of Evolutionary Multiobjective Optimization Techniques: State of the Art and Future Trends. In *Proc. of the Congress on Evolutionary Computation*, pages 3–13, 1999.

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

[5] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 1–11, 2000.

[6] Kalyanmoy Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley, 2001.

[7] Levente Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. *Euro Conference in Machine Learning*, 4212:282–293, 2006.

[8] Tomas Kozelek. *Methods of MCTS and the game Arimaa*. M.s. thesis, Charles Univ., Prague, 2009.

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

[10] Diego Perez, Edward J. Powley, Daniel Whitehouse, Philipp Rohlfshagen, Spyridon Samothrakis, Peter I. Cowling, and Simon Lucas. Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions. *IEEE Transactions on Computational Intelligence and AI in Games*, (submitted):to appear, 2013.

[11] Diego Perez, Philipp Rohlfshagen, and Simon Lucas. The Physical Travelling Salesman Problem: WCCI 2012 Competition. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2012.

[12] Diego Perez, Spyridon Samothrakis, Simon Lucas, and Philipp Rohlfshagen. Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games. In *Proc. of the Conference on Genetic and Evolutionary Computation (GECCO)*, page to appear., 2013.

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

[14] Peter Vamplew, Richard Dazeley, Adam Berry, Rustam Issabekov, and Evan Dekker. Empirical Evaluation Methods for Multiobjective Reinforcement Learning Algorithms. *Machine Learning*, 84:51–80, 2010.

[15] Wang Weijia and Michele Sebag. Multi-objective Monte Carlo Tree Search. In *Proceedings of the Asian Conference on Machine Learning*, pages 507–522, 2012.

[16] Aimin Zhou, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. Multiobjective Evolutionary Algorithms: A Aurvey of the State of the Art. *Swarm and Evolutionary Computation*, 1:32–49, 2011.