Chapter 3 - Planning in GVGAI

Diego Perez-Liebana and Raluca D. Gaina

1 Introduction

This chapter focuses on AI methods used to tackle the planning problems generated by GVGAI. Planning problems refer to creating plans of action so as to solve a given problem; for example, figuring out how to first pick up a key and use it to open the exit door of a level while avoiding enemies, in an adventure game like Zelda. For this purpose, a model of the environment is available to simulate possible future states, when given a current state and an action to be taken by the player; this model will be referred to as a Forward Model (FM) from now on. Not all modern and complex games may have an FM available, an issue addressed in the Learning track of the framework, see Chapter 5. However, using the game engine to not only run the game, but also allow AI players to simulate states internally, is possible in many games and inherent to human intuition and processing of different scenarios. Creating a plan, imagining what may happen if that plan is executed and figuring out the best course of action is one method for human decision making which is widely applicable to a range of problems. Therefore, landing this ability to AI to reason in interesting scenarios, such as those depicted by GVGAI games, would take the technology one step closer to general intelligence.

In this chapter we explore several methods which apply and extend on this theory. Monte Carlo Tree Search (MCTS) and Rolling Horizon Evolutionary Algorithms (RHEA) are the foundations on which most planning AI agents are built, featuring two different approaches: the first builds a game tree out of possible game actions and game states, from which it extracts statistics to decide what to do next in any given state; the second creates entire plans and uses evolutionary computation to combine and modify them so as to end up with the best option to execute in the game. These basic methods can be extended in several ways: knowledge learned about the environment can be imbued into the algorithm to better inform action plan evaluation, as well as other heuristics which allow the AI to make better than random decisions in the absence of extrinsic reward signals from the game. Additionally, GVGAI problems can be seen as multi-objective optimisation problems: the scoring systems in the game may be deceptive and the AI would need to not only focus on gaining score (as in traditional arcade games), but also on solving the problem and winning the game, possibly even considering time constraints. Furthermore, the extensive research in Evolutionary Algorithms can be applied to RHEA methods for better results, for example by seeding the initial population with better than random action plans, by keeping a statistical tree similar to the approach taken by MCTS, by using statistics for modifying action plans, by keeping action plans in-between game ticks instead of starting from scratch every time or by dynamically modifying the action plan length depending on features of the current game state (i.e. exploring further ahead rewards vs. gathering enough statistics about close rewards to make informed decisions).

All of these enhancements are described in detail in the following sections of this chapter. In all experiments we compare the enhanced algorithms performance against the vanilla version as well as against other methods (i.e. RHEA vs MCTS) to explore whether the modifications proposed improve upon the basic algorithm, as well as whether the new algorithm is able to compete with the state of the art and obtain better performance. The methods are tested on a subset of GVGAI games selected for the particular experiments, with the aim of either tackling specific game types or obtaining a diverse set of games proposing different types of challenges. All agents have $40ms$ of decision time in the competition setting, and the works described here have normally respected this budget. In some cases, due to testing performed in multiple machines, an architecture-independent approach has been followed to avoid undesired bias in the experiments: instead of using time as budget, a maximum number of calls to the advance function of the FM is established.

This chapter is divided into four sections. Section 2 describes Monte Carlo Tree Search (MCTS), followed by two sections showing some interesting variants for GVGAI (Knowledge-based and Multi-objective MCTS in sections 3 and 4, respectively). Finally, Section 5 explores the use of Rolling Horizon Evolutionary Algorithms (RHEA) and several enhancements.

2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search algorithm initially proposed in 2006 [3,4,16]. Originally, it was applied to board games and is closely associated with Computer Go, where it led to a breakthrough in performance [17] before the irruption of deep learning and AlphaGo. MCTS is especially will suited for large branching factor games like Go, and this lead to MCTS being the first algorithm able to reach professional level play in the 9×9 board size version [10]. MCTS rapidly became popular due to its significant success in this game, where traditional approaches had been failing to outplay experienced human players.

MCTS has since then been applied to a wide range of games, including games of hidden information, single-player games and real-time games. MCTS has also been used extensively in general game playing (GGP) [5] with very good results. An extensive survey that includes a description of MCTS, a complete collection of MCTS variants, and multiple applications of this algorithm to games and other domains, can be found at [2].

The algorithm builds a search tree that grows in an asymmetric manner by adding a single node at a time, estimating its game-theoretic value by using self-play from the state of the node to the end of the game. Each node in the tree keeps certain statistics that indicate the empirical average ($Q(s, a)$) of the rewards obtained when choosing action a at state s, how often a move a is played from a given state s ($N(s, a)$) and how many times a state s has been visited ($N(s)$). The algorithm builds the tree in successive iterations by simulating actions in the game, making move choices based on these statistics.

Each iteration of MCTS is composed of several steps [11]: *Tree selection*, *Expansion*, *Simulation* and *Backpropagation*. Figure 1 depicts these four steps in the algorithm.

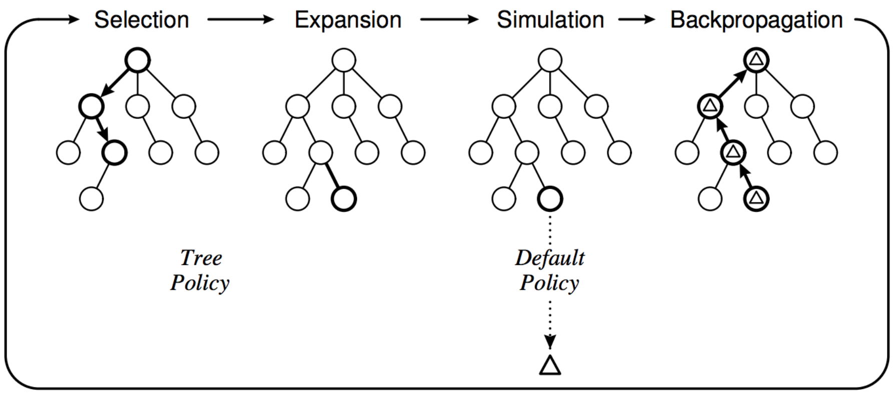

Fig. 1: MCTS algorithm steps [2]: Selection, Expansion, Simulation and Backpropagation are executed iteratively until the allocated budget expires (time, iterations, or uses of the Forward Model.)

At the start, the tree is composed only of the root node, which is a representation of the current state of the game. During the *selection* step, the tree is navigated from the root until a maximum depth or the end of the game has been reached. In this stage, actions are selected using a *multi-armed bandit* policy (tree policy) and applying it to the FM.

Multi-armed bandits is a known problem that features a slot machine with multiple arms. When an arm is pulled, a reward r is received drawn from an unknown probability distribution. The objective is to minimise the regret (or maximise the cumulative reward) when pulling from the different arms in sequence. In this scenario, regret is defined as the opportunity loss when choosing a sub-optimal arm. Good policies in this problem select actions by balancing the exploration of available arms and the exploitation of those that provided better rewards in the past. Auer et al. [1] proposed the *Upper Confidence Bound* (UCB1, see Equation 1) policy for arm bandit selection, and Kocsis and Szepesvári applied it later for tree search (UCT: Upper Confidence Bound for trees) [16].

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

The objective is to find an action a that maximizes the value given by the UCB1 equation. $Q(s,a)$ is the *exploitation* term, while the second term (weighted by the constant C) is the *exploration* term. The exploration term relates to how many times each action a has been selected from the given state s, and the amount of selections taken from the current state ($N(s,a)$ and $N(s)$, respectively). The value of C balances between exploration and exploitation. If the balancing weight $C = 0$, UCB1 behaves greedily following the action with the highest average outcome so far. If rewards $Q(s,a)$ are normalized in the range $[0,1]$, a commonly used value for C in single-player games is $\sqrt{2}$. The optimal value of C may vary from game to game.

The *tree selection* phase continues navigating the tree until a node with fewer children than the available number of actions is found. Then, a new node is added as a child of the current one (*expansion* phase) and the *simulation* step starts. From the new node, 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 an end-game state (or a pre-defined depth) is reached, where the state of the game is evaluated. Finally, during the *backpropagation* step, the statistics $N(s)$, $N(s,a)$ and $Q(s,a)$ are updated for each node traversed, using the reward obtained in the evaluation of the state. These steps are executed in a loop until a termination criteria is met (such as number of iterations, or when the time budget is consumed).

Once all iterations have been performed, MCTS recommends an action for the agent to take in the game. This recommendation policy determines an action in function of the statistics stored in the root node. For instance, it could return the

Algorithm 1 General MCTS approach [2].

Input: Current Game State s_0
Output: Action to execute by the AI agent this turn.

```
1:  function MCTS_Search($s_0$)
2:      create root node $v_0$ with state $s_0$
3:      while within computational budget do
4:          $v_l$ = TreePolicy(($v_0$))
5:          $\Delta$ = DefaultPolicy(($s(v_l)$))          ▷ $s(v_l)$: State in node $v_l$, $\Delta$: reward for state $s$.
6:          Backup(($v_l$, $\Delta$))
7:      return $a$(BestChild(($v_0$)))
8:
9:  function TreePolicy($v$)
10:     while $v$ is nonterminal do
11:         if($v$ not fully expanded)
12:         return Expand(($v$))
13:         else
14:         $v \leftarrow$ UCB1(($v$))          ▷ Eq. 1.
15:         $s \leftarrow f(v(s), a(v))$
16:     return $v$
17:
18: function Expand($v$)
19:     choose $a \in$ untried actions from $A(s(v))$          ▷ $A(s(v))$: Available actions from state $s(v)$.
20:     add a new child $v'$ to $v$
21:     with $s(v') = f(s(v), a)$          ▷ $f(s(v), a)$: State reached from $s(v)$ after applying $a$.
22:     return $v'$
23:
24: function DefaultPolicy($s$)
25:     while $s$ is non-terminal do
26:         choose $a \in A(s)$ uniformly at random
27:         $s \leftarrow f(s, a)$
28:     return reward for state $s$
29:
30: function Backup($v, \Delta$)
31:     while $v$ is not null do
32:         $N(s(v)) \leftarrow N(s(v)) + 1$
33:         $N(s(v), a(v)) \leftarrow N(s(v), a(v)) + 1$          ▷ $a(v)$: Last action applied from state $s(v)$.
34:         $Q(s(v), a(v)) \leftarrow Q(s(v), a(v)) + \Delta$
35:         $v \leftarrow$ parent of $v$
```

action chosen more often (a with the highest $N(s, a)$), the one that provides a highest average reward ($argmax_{a \in A(s)}$ $Q(s, a)$), or simply to apply Equation 1 at the root node. The pseudocode of MCTS is shown in Algorithm 1.

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.

The GVGAI Framework provides a simple implementation of MCTS (also referred to here as *sampleMCTS* or *vanilla MCTS*). In the rest of this book, we refer to basic methods as "vanilla", i.e. methods containing the bare minimum steps for the algorithm, without any enhancements or tailoring for efficiency, performance or specific cases. In this MCTS implementation, C takes a value of $\sqrt{2}$ and rollout depth is set to 10 actions from the root node. Each state reached at the end of the *simulation* phase is evaluated using the score of the game at that point, normalized between the minimum and maximum scores ever seen during the play-outs. If the state is terminal, the reward assigned is a large positive (if the game is won) or negative (in case is lost) number.

The vanilla MCTS agent performs relatively well in GVGAI games[1], but with obvious limitations given that there is no game-dependent knowledge embedded in the algorithm. The value function described above is based exclusively in score and game end state, concepts that are present in all games and thus general to be used in a GVGP method. The following two sections propose two modifications to the standard MCTS to deal with this problem. The first one, Knowledge-based Fast Evolutionary MCTS (Section 3), explores the modification of the method to learn from the environment while the game is being played. The second one, proposes a Multi-Objective version of MCTS (Section 4) for GVGP.

3 Knowledge-based Fast Evolutionary MCTS

3.1 Fast Evolution in MCTS

Knowledge-based Fast Evolutionary MCTS (KB Fast-Evo MCTS) is an adaptation for GVGAI of a previous work by [21], who proposed an MCTS approach that uses evolution to learn a rollout policy from the environment. Fast Evolutionary MCTS uses evolution to adjust a set of weights w to bias the Monte Carlo simulations. These weights are used to select an action at each step in combination with a fixed set of features extracted for the current game state.

[1] See actual results in upcoming sections.

Each rollout evaluates a single individual (set of weights), using the value of the state reached at the end as the fitness. The evolutionary algorithm used to evolve these weights is a $(1+1)$ Evolution Strategy (ES). The pseudocode of this algorithm can be seen in Algorithm 2.

Algorithm 2 Fast Evolutionary MCTS Algorithm, from [21], assuming one roll-out per fitness evaluation.

Input: v_0 root state.
Output: weight vector w, action a.

```
1: while within computational budget do
2:     $w = \text{Evo.GetNext}()$
3:     Initialize Statistics Object $S$
4:     $v_l = \text{TreePolicy}((v_0))$
5:     $\delta = \text{DefaultPolicy}((s(v_l), D(w)))$
6:     $\text{UpdateStats}((S, \delta))$
7:     $\text{Evo.SetFitness}((w, S))$
8: return $w = \text{Evo.getBest}()$, $a = \text{recommend}((v_0))$
```

The call in line 3 retrieves the next individual to evaluate (w), a and the fitness is set in line 8. The vector of weights w is used to bias the rollout (line 6). For each state found in the rollout, first a number of N features are extracted (mapping from state space S to feature space F). Given a set of available actions A, a weighted sum of feature values determines the relative strength of each action (a_i), as shown in Equation 2.

Given that actions are weighted per feature, all weights are stored in a matrix W, where each entry w_{ij} is the weighting of feature j for action i. A softmax function (see Equation 3) is used to select an action for the Monte Carlo simulation.

$$a_i = \sum_{j=1}^{N} w_{ij} \times f_i; \tag{2}$$

$$P(a_i) = \frac{e^{-a_i}}{\sum_{j=1}^{A} e^{-a_j}} \tag{3}$$

The features selected to bias this action selection are euclidean distances from the avatar to the closest NPC, resource, non-static object and portal. In GVGAI, it is not possible to determine a feature space a priori (as the same method should work for any, even unknown, game). In fact, given that these features depend on the existence of certain sprites that may appear or disappear mid-game, the number of N features does not only vary from game to game, but also varies from game step to game step. Therefore, the algorithm needs to be flexible to adapt the vector of weights to the number of features at each state.

3.2 Learning Domain Knowledge

The next step in defining KB Fast-Evo MCTS is to add a system that can provide a stronger fitness function for the individuals being evolved. Given that this fitness is calculated at the end of a rollout, the objective is to define a state evaluation function with *knowledge items* learnt dynamically when playing the game.

In order to build this function, we define *knowledge base* as the combination of two factors: *curiosity* plus *experience*. For this work, *curiosity* refers to the discovery of the consequences of colliding with other sprites, while *experience* weights those events that provided a score gain. In both cases, the events logged are those where the avatar, or a sprite produced by the avatar, collides with another object in the game for which a feature is extracted (NPC, resource, non-static object and portal). Each one of these *knowledge items* keeps the following statistics.

- Z_i: number of occurrences of the event i.
- $\overline{x_i}$: average of the score *change*, which is the difference between the game score before and after the event took place. As GVGAI games are quite dynamic, multiple events happen at the same game tick and it is not possible to certainly assess which event actually triggered the score change. Therefore, the larger the number of occurrences Z_i, the more precise $\overline{x_i}$ will be.

These two values are updated after every use of the FM in the Monte Carlo simulation. When the state at the end of the rollout is reached, the following values are computed.

- Score change ΔR: this is the difference in game score between the initial and final states of the rollout.
- *Curiosity:* Knowledge change $\Delta Z = \sum_{i=1}^{N} \Delta(K_i)$, which measures the change of all Z_i in the knowledge base, for each knowledge item i. $\Delta(K_i)$ is calculated as shown in Equation 4, where Z_{i0} is the value of Z_i at the beginning of the rollout and Z_{iF} is the value of Z_i at the end.

$$\Delta(K_i) = \begin{cases} Z_{iF} & : Z_{i0} = 0 \\ \frac{Z_{iF}}{Z_{i0}} - 1 & : \text{Otherwise} \end{cases} \tag{4}$$

$\mathit{\Delta} Z$ is higher when the rollouts produce more events. Events that have been rarely triggered before will provide higher values of $\mathit{\Delta} Z$, favouring knowledge gathering in the simulations.

– *Experience:* $\mathit{\Delta} D = \sum_{i=1}^{N} \mathit{\Delta}(D_i)$. This is a measure of change in the distance to each sprite of type i from the beginning to the end of the rollout. Equation 5 defines the value of $\mathit{\Delta}(D_i)$, where D_{i0} is the distance to the closest sprite of type i at the beginning of the rollout, and D_{iF} is the same distance at the end of the rollout.

$$\mathit{\Delta}(D_i) = \begin{cases} 1 - \frac{D_{iF}}{D_{i0}} & : Z_{i0} = 0 \text{ OR} \\ & \quad D_{i0} > 0 \text{ and } \overline{x_i} > 0 \\ 0 & : \text{Otherwise} \end{cases} \tag{5}$$

Note that $\mathit{\Delta} D$ will be higher if the avatar reduced the distance to those sprites with a positive $\overline{x_i}$ (i.e. provided a bost in score in the past) during the rollout, apart from reducing the distance to unknown sprites.

Equation 6 describes the final value for the game state and fitness for the individual being evaluated. This value is $\mathit{\Delta} R$, unless $\mathit{\Delta} R = 0$. When $\mathit{\Delta} R = 0$, none of the actions during the rollout changed the score of the game, so the value must reflect the curiosity and experience components. This is done via a linear combination with weights $\alpha = 0.66$ and $\beta = 0.33$.

$$Reward = \begin{cases} \mathit{\Delta} R & : \mathit{\Delta} R \neq 0 \\ \alpha \times \mathit{\Delta} Z + \beta \times \mathit{\Delta} D & : \text{Otherwise} \end{cases} \tag{6}$$

Therefore, the new value function gives priority to actions that produce a score gain. However, when no score gain is achieved, actions that provide more information to the knowledge base or get the avatar closer to sprites that provide score are awarded.

3.3 Experimental work

Four different configurations of KB Fast-Evo MCTS have been tested on the 10 games of the first set of GVGAI games. Each game has 5 levels and each level has been played 5 times, totalling 250 games played for each one of the following configurations.

– **Vanilla MCTS**: the sample MCTS implementation from the framework, at the beginning of this section.

- **Fast-Evo MCTS**: Fast Evolutionary MCTS, as per Lucas et al. [21] (adapted to use a dynamic number of features).
- **KB MCTS**: Knowledge-based (KB) MCTS as explained in the previous section, but using uniformly random selection in the rollouts (i.e., no evolution biases the Monte Carlo simulations).
- **KB Fast-Evo MCTS**: Version of MCTS that uses both the knowledge base and evolution to bias the rollouts.

Performance in GVGAI is typically measured in two ways: percentage of games won and average score achieved in them. It's worthwhile highlighting that the former measure is easier to use for comparisons than the later, as each game has a different scoring system with different bounds and profiles.

Table 1 shows the win percentages achieved by each one of the four algorithm in the games used for testing. If we observe the total average of victories, *KB Fast-Evo MCTS* leads the comparison with 49.2% of games won. All the other variants achieved rates between 30.8% and 33.2%, showing that the addition of both the knowledge base and the fast evolution of weights to bias rollouts provides a boost in performance. However, the addition of any of them separately does not impact the algorithm significantly.

Game	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**
Aliens	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)	100.0 (0.0)
Boulderdash	0.0 (0.0)	4.0 (3.9)	**28.0 (9.0)**	**16.0 (7.3)**
Butterflies	88.0 (6.5)	**96.0 (3.9)**	80.0 (8.0)	**100.0 (0.0)**
Chase	12.0 (6.5)	12.0 (6.5)	0.0 (0.0)	**92.0 (5.4)**
Frogs	24.0 (8.5)	16.0 (7.3)	8.0 (5.4)	20.0 (8.0)
Missile Command	20.0 (8.0)	20.0 (8.0)	20.0 (8.0)	**56.0 (9.9)**
Portals	12.0 (6.5)	**28.0 (9.0)**	16.0 (7.3)	**28.0 (9.0)**
Sokoban	0.0 (0.0)	0.0 (0.0)	**4.0 (3.9)**	**8.0 (5.4)**
Survive Zombies	44.0 (9.9)	36.0 (9.6)	52.0 (10.0)	44.0 (9.9)
Zelda	8.0 (5.4)	**20.0 (8.0)**	8.0 (5.4)	**28.0 (9.0)**
Overall	30.8 (2.6)	33.2 (2.7)	31.6 (2.6)	**49.2 (3.2)**

Table 1: Percentage of victories obtained on each game, standard error between parenthesis. In bold, those results that are the best ones on each game. Each value corresponds to the average result obtained by playing that particular game 25 times.

Looking at individual games, it can be seen that, in most games, *KB Fast-Evo MCTS* outperforms *Vanilla MCTS* in victory rate. In some games, the improvement

Game	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**
Aliens	36.72 (0.9)	38.4 (0.8)	37.56 (1.0)	**54.92 (1.6)**
Boulderdash	9.96 (1.0)	12.16 (1.2)	**17.28 (1.7**	**16.44 (1.8)**
Butterflies	27.84 (2.8)	31.36 (3.4)	31.04 (3.4)	28.96 (2.8)
Chase	4.04 (0.6)	4.8 (0.6)	3.56 (0.7)	**9.28 (0.5)**
Frogs	-0.88 (0.3)	-1.04 (0.2)	-1.2 (0.2)	-0.68 (0.2)
Missile Command	-1.44 (0.3)	-1.44 (0.3)	-1.28 (0.3)	**3.24 (1.3)**
Portals	0.12 (0.06)	**0.28 (0.09)**	0.16 (0.07)	**0.28 (0.09)**
Sokoban	0.16 (0.1)	0.32 (0.1)	**0.7 (0.2)**	**0.6 (0.1)**
Survive Zombies	13.28 (2.3)	14.32 (2.4)	18.56 (3.1)	21.36 (3.3)
Zelda	0.08 (0.3)	0.6 (0.3)	0.8 (0.3)	0.6 (0.3)
Overall	9.0 (0.9)	10.0 (1.0)	10.7 (1.0)	**13.5 (1.2)**

Table 2: Scores achieved on each game, standard error between parenthesis. In bold, those results that are the best ones on each game. Each value corresponds to the average result obtained by playing that particular game 25 times.

can be observed as a consequence of adding either a stronger evaluation function or the fast evolution component (*Boulderdash* or *Zelda*, respectively). Although the addition of both parts also drives the improvement in other games (such as *Missile Command* and *Chase*).

Table 2 shows the average scores achieved by the same methods in these games. As in the case of the victory rate, *KB Fast-Evo MCTS* also gets higher total average score than the other methods, with 13.5 ± 1.2 points versus the range 9 to 11 achieved by the other algorithms.

As mentioned above, it is more relevant to compare scores on a game by game basis due to the different score systems employed per game. In this case, *KB Fast-Evo MCTS* still outperforms *Vanilla MCTS* in most games in scores.

It can also be observed in the results that *KB Fast-Evo MCTS* fails to provide good results in certain games. Games like *Sokoban* and *Frogs* show little or no improvement at all. We hypothesize that the reasons for this are varied. One of them is the use of Euclidean distances for feature extraction. Using distances calculated by a path-finding algorithm such as A* would be more accurate, but the impact in the real-time nature of the algorithm would be quite high. Additionally, path-finding requires the definition of a navigable space, which is hard to describe as a general concept for *any* game.

However, it is likely that the major struggle is the relevance of the features taken for some of the games. For instance, in *Sokoban*, the avatar must push boxes to win the game and the orientation in which the box is pushed (and where it is pushed from) is relevant. But the features used here do not capture this information, as all collision events triggered when colliding with a box are treated the same. In other cases, like in *Frogs*, a specific sequence of actions must be applied so the avatar crosses the road without being hit by a truck. However, these sequences are rare - in most cases they collide with trucks, which ends in a game loss. Therefore the algorithm rewards not getting closer to these trucks and staying safe without incentives to cross the road.

4 Multi-Objective MCTS for GVGAI

This section proposes another modification to the MCTS algorithm for real-time games in which state evaluations consider two objectives instead of one. In GVGAI, these objectives are to maximize game score (same objective as in the vanilla MTCS described in Section 2) and to provide an incentive to maximize exploration in the game level. First, we provide some background in Multi-objective Optimization, to then describe the Multi-Objective MCTS (MO-MCTS) approach and the experimental work carried out to test this new algorithm.

4.1 Multi-objective Optimization

An optimization problem is called multi-objective when two or more conflicting objective functions must be optimized simultaneously. A multi-objective optimization problem can be defined as:

$$\textit{optimize} \quad \{f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_m(\boldsymbol{x})\} \tag{7}$$

subject to $\boldsymbol{x} \in \Omega$, with $m(\geq 2)$ conflicting objective functions $f_i : \Re^n \rightarrow \Re$. $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)^T$ are *decision vectors* from the feasible region $\Omega \subset \Re^n$. $Z \subset \Re^m$ is the feasible objective region. The elements of this region are known as *objective vectors*, which consist of m objective values $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \cdots, f_m(\boldsymbol{x}))$. Each solution $\boldsymbol{x}$ results in a set of m different values to be optimized.

One solution $\boldsymbol{x}$ is said to *dominate* another solution $\boldsymbol{y}$ if and only if:

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

If these two conditions are met, it is said that $\boldsymbol{x} \prec \boldsymbol{y}$ ($\boldsymbol{x}$ *dominates* $\boldsymbol{y}$). This condition determines a *partial ordering* between solutions in the objective space. In

Algorithm 3 Node update in the backpropagation step of MO-MCTS [24].

Input: *node* current tree node being updated
Input: $\overline{r}$ reward of the last state evaluation
Input: *dominated* if $\overline{r}$ is dominated by a front from the descendants of *node*

```
1: function UPDATE($node, \overline{r}, dominated = false$)
2:     $node.Visits = node.Visits + 1$
3:     $node.\overline{R} = node.\overline{R} + \overline{r}$ $!dominated$ $node.P \prec \overline{r}$
4:     $dominated = true$
5:     $node.P = node.P \cup \overline{r}$          ▷ $P$ is the Pareto front approximation at each node
6:     UPDATE(($node.parent, \overline{r}, dominated$))
```

the case where it is not possible to state that $\boldsymbol{x} \prec \boldsymbol{y}$ or $\boldsymbol{y} \prec \boldsymbol{x}$., it is said that these solutions are *indifferent* to each other. Solutions indifferent to each other form part of the same *non-dominated set.* A non-dominated set P is said to be the *Pareto-set* if there is no other solution in the decision space that dominates any member of P. The objective vectors of P build a *Pareto-front.*

One of the most popular methods to measure the quality of a non-dominated set is the Hypervolume Indicator (HV). This indicator measures both the diversity and convergence of non-dominated solutions [29]. The HV of a Pareto front P ($HV(P)$) is defined as the volume of the objective space dominated by P. The higher the value of $HV(P)$ the better the front is, assuming all objectives are to be maximized.

4.2 Multi-objective MCTS

Multi-Objective Monte Carlo Tree Search (MO-MCTS) [24] tackles the problem of selecting an action with a reduced time budget in an MO setting. The algorithm requires that a game state is evaluated according to m objectives, returning a vector $\overline{r}$. $\overline{r}$ is the reward vector to be used in the *backpropagation* step of MCTS through all the nodes visited in the last iteration. This vector updates an accumulated reward vector $\overline{R}$ and a local Pareto front approximation P at each node. Algorithm 3 describes how the update of the node statistics in MO-MCTS.

The Pareto front P update (line 5 of Algorithm 3) in a node works as follows: in the case that $\overline{r}$ is *not* dominated by the front, $\overline{r}$ is added to P. This update considers that some of the current points in P may leave the set if $\overline{r}$ dominates any of them. If $\overline{r}$ is dominated by P, the front remains unchanged. Note that, if the latter is true, no further fronts will be changed in all remaining nodes until reaching the root.

Each node can have an estimation over the quality of the reachable states by keeping its local front P updated. The quality of the front is calculated as the hypervolume $HV(P)$, which substitutes the exploitation term $Q(s,a)$ in the MO-UCB equation (8).

$$a^* = \operatorname*{arg\,max}_{a \in A(s)} \left\{ HV(P) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{8}$$

As the backpropagation step is followed until reaching the root node, it is straightforward to see that the root contains the best non-dominated front of the whole search. Therefore, the root note can also provide information to the recommendation policy to select an action to play in the game once the budget is over. In MO-MCTS, the root stores information about which action leads to which point in its own non-dominated front. Weights can then be defined to determine which point of the front P at the root is chosen and pick the action that leads to it.

Heuristics for GVGAI This section proposes two heuristics for GVGAI methods, each one of them will be treated as objectives in the experimental testing for this approach.

Score (Objective O_1) This is the simple heuristic describe above that uses on the game score and the game end condition: the value of a state is the current score unless the game is over. In that case, a large integer is added if the game is won (10^6) or lost (-10^6).

Level Exploration (Objective O_2) This heuristic computes a value that rewards an agent that maximizes the number of grid cells visited in the level. Its implementation is based on *pheromone* trails and it was first proposed in [26]. The technique works as follows: the mechanism simulates that the avatar expels pheromones at each game tick, which spread into nearby cells. Each cell holds a pheromone value $p_{ij} \in [0,1]$, where i and j are coordinates in the level grid. p_{ij} is set to decay with time, and the change of pheromone p_{ij} between steps is given by Equation 9.

$$p_{i,j} = \rho_{df} \times \rho_{\phi} + (1 - \rho_{df}) \times \rho_{dc} \times p_{i,j} \tag{9}$$

ρ_{ϕ} is computed as the sum of pheromone trail in all surrounding cells divided by the number of neighbouring cells[2]. $\rho_{df} \in (0,1)$ sets the value of pheromone diffusion and $\rho_{dc} \in (0,1)$ establishes the decay rate of the pheromone value at each frame. Values that showed good results previously are $\rho_{df} = 0.4$ and $\rho_{dc} = 0.99$.

[2] An edge cell has fewer neighbours.

This algorithm produces high values of pheromone trail in the close proximity of the avatar, as well as in positions recently visited. Figure 2 shows an example of how much pheromone is added to a cell and its neighbours.

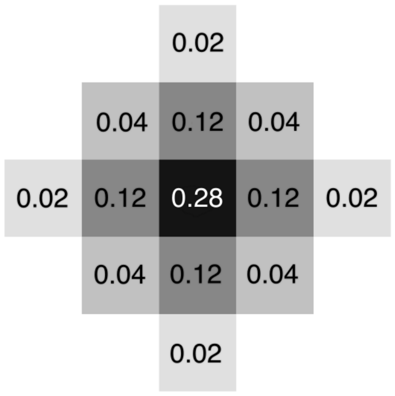

Fig. 2: Pheromone diffusion [27]. The avatar is assumed to be located in the central position, where more pheromone is deposited. The neighbouring cells receive less pheromone. These values are added to the pheromone previously deposited on each cell (values always kept $\in (0, 1)$.

This heuristic values higher positions where the pheromone value is small, which rewards agents that maximize the exploration of the level. Thus, the value of the heuristic is computed as $O_2 = 1 - p_{i,j}$, where i and j are the coordinates of the avatar in the grid. As in the previous case, a large positive or negative number is added to O_2 if the game is won or lost in a state - otherwise the agent would miss the opportunity of winning the game or, even worse, lose it in favour of exploring new positions.

4.3 Experimental Work

Four different MCTS approaches are tested in this study. All these algorithms share the value of $C = \sqrt{2}$ for the UCB1/MO-UCB Equations (1/8), a limit of 50 iterations per game tick and a simulation depth of 10 moves from the root node. The experiments were conducted in the games of the first GVGAI set (10 games). Each approach described above played 100 times each one of the 5 levels, leading to 500 repetitions per game and agent. These approaches are listed next.

– Sample MCTS. The sample MCTS controller included in the GVGAI framework. The state value function is exactly O_1 in this case.

- Weighted Sum MCTS. A classical alternative to multi-objective approaches is to combine all the objectives into a linear combination. This implementation uses the default MCTS, but the state value function is determined as $O_1 \times \alpha + O_2 \times \beta$, with $\alpha = \beta = 0.5$.
- Mixed Strategy MCTS. The idea of this approach is to tackle the multi-objective problem as a mixed strategy [22]. In this setting, each objective O_1, O_2 is managed by a different state evaluation function. The algorithm used is still sample MCTS but, at the beginning of the decision time, a higher-level policy determines which objective should be considered during that frame. In the experiments described here, the selection of one or the other is done uniformly at random.
- MO-MCTS. This approach implements the MO-MCTS algorithm described above, using O_1 and O_2 as the two objectives to optimize. The recommendation policy uses a linear combination $O_1 \times \alpha + O_2 \times \beta$ ($\alpha = \beta = 0.5$ as in the previous approaches) to evaluate each member of the Pareto front P owned by the root. The action that leads to the point with the highest weighed sum is picked to be played in the game.

Figure 3 shows the win rate of all agents in the games tested. It shows Sample MCTS and MO-MCTS as the strongest agents of the four, either of them achieving the highest percentage of victories on each game. MO-MCTS outperforms Sample MCTS in five games (*Aliens*, *Frogs*, *Missile Command*, *Portals* and *Zelda*).

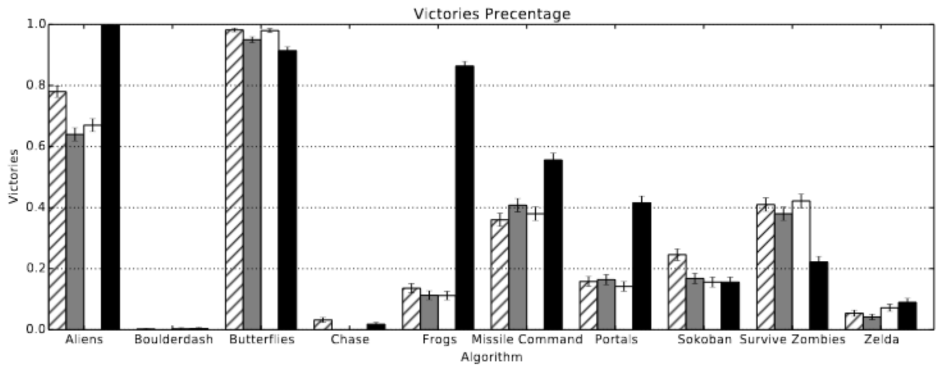

Fig. 3: Percentage of victories (with std. error). Four approaches are compared per game. From left to right: Sample MCTS, Weighted Sum MCTS, Mixed Strategy MCTS, MO-MCTS. From [27].

MO-MCTS also achieves the highest win percentage if compared across all games to the other approaches. Table 3 shows this comparison, which indicates that Sample MCTS is the second best algorithm in this ranking.

	Sample MCTS	**Weighted Sum MCTS**	**Mixed Strategy MCTS**	**MO-MCTS**
% Victories	32.24 (0.67)	29.80 (0.66)	30.51 (0.66)	**42.38 (0.70)**

Table 3: Win rate achieved across all games in the first GVGAI game set.

Some interesting observations can be made when analyzing the results in a game per game basis. *Aliens*, for example, is a game that has traditionally been well played by Sample MCTS, achieving 100% of victories. This study uses 50 iterations per method, which approximately halves the budget that $40ms$ per tick allows for this game in particular. With this reduction, the performance of Sample MCTS drops, while MO-MCTS achieves a $99,80\%(0.20)$ win rate. Another impressive result is the high victory rate achieved by MO-MCTS in *Frogs*. This game has always posed many problems to MCTS approaches (see, for instance, Table 1 from section 2), but MO-MCTS agent uses the exploration heuristic to find the goal of the level in 86.40% (1.53) of the games played.

In some cases MO-MCTS achieves worse result than the other algorithms which seems to indicate that the excessive exploration can be a disadvantage. An example of this is *Survive Zombies*. In this game, a good strategy is to find a spot save from zombies and stay there. An excessive amount of exploration may lead to encounter more enemies and lose the game.

Not all games are favorable to MO-MCTS, however, and in some cases it is possible that the excessive exploration is actually a disadvantage. A good example could be *Survive Zombies*, where one of the best strategies is to locate a spot in the level safe from zombies. Exploring the level too much may lead to find more enemies and therefore to lose the game.

Figures 4 and 5 show the average of scores achieved by all approaches in the 10 games. Attending to this metric, MO-MCTS achieves the highest score in 7 out of 10 games, only beaten or matched by Sample MCTS on the other 3.

Table 4 shows the numerical data of these results.

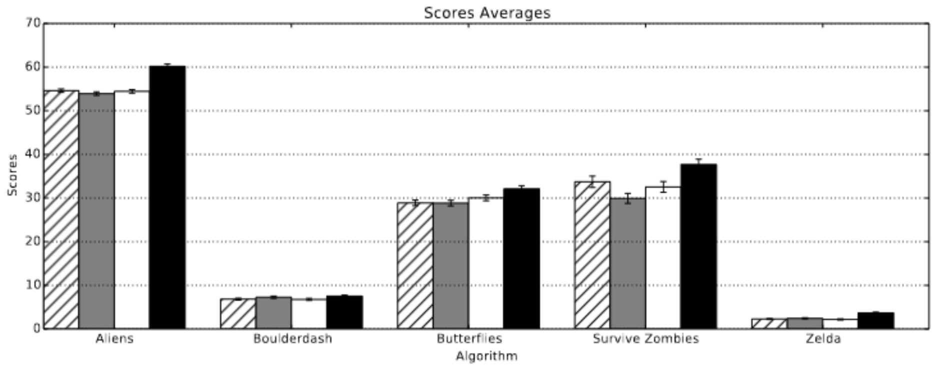

Fig. 4: Average of scores (with std. error). Four approaches are compared per game. From left to right: Sample MCTS, Weighted Sum MCTS, Mixed Strategy MCTS, MO-MCTS. From [27].

MO-MCTS is significantly better in terms of scores than all approaches in seven games. It is worthwhile to highlight that MO-MCTS behaves clearly better (both in victories and in score) than the other multi-objective variants. These results seem to suggest that using multiple objectives to tackle GVGAI is promising, but the way these objectives are used is decisive to achieve good results. Combining them in a linear combination or using them in an alternative manner does not produce as good results as using the Pareto fronts in MO-MCTS.

Mixed Strategy MCTS does not achieve good results in this study, but a closer look at them having in mind the way this algorithm operates suggests some interesting insights. This controller spends 50% of its moves only focusing on new places to move to, without considering the score, and when this happens is determined at random. This is the only agent studied here that completely ignores the score in half of its moves. However, Mixed Strategy MCTS is significantly better than Weighted-Sum MCTS in two games (precisely in terms of score) and no worse in the other eight. This suggests that mixed strategies can work well in GVGP if objectives are chosen at the right time, encouraging further investigation on better (i.e. probably dynamic) balancing of the different objectives while playing.

5 Rolling Horizon Evolutionary Algorithms

Rolling Horizon Evolutionary Algorithms (RHEA) were first introduced by Perez et al. [23] as an alternative (and potentially better, more adaptive option) to MCTS for online planning in games. The concept they put forward was a novel usage of EAs

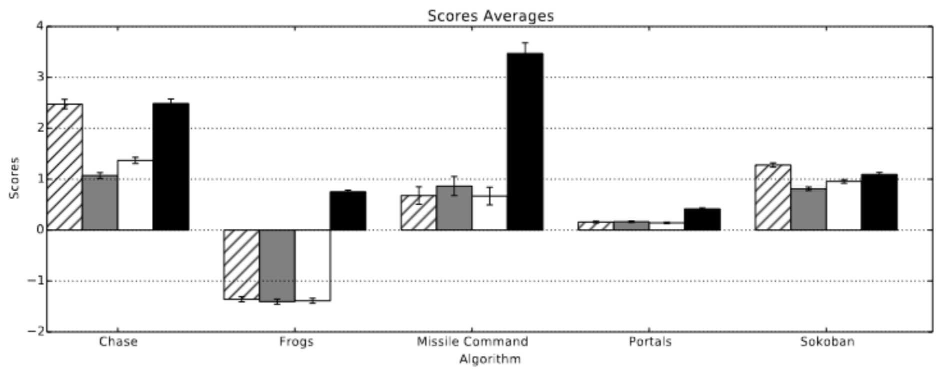

Fig. 5: Average of scores (with std. error). Four approaches are compared per game. From left to right: Sample MCTS, Weighted Sum MCTS, Mixed Strategy MCTS, MO-MCTS. From [27].

for optimisation of sequences (or plans) of actions in games. Therefore, the solution evolved by the EA is an action sequence of a specific length, which is executed for evaluation using simulations of a model of the game. This could be seen as similar to an MCTS rollout, where the final state reached after advancing through the actions in an individual is evaluated and gives its fitness value.

This technique has slowly become more popular in game AI research. Several authors applied RHEA to specific games, starting from the single-player real-time Physical Travelling Salesman Problem in 2013 [23], to a two-player real-time Space Battle game in 2016 [18], or single-player real-time games Asteroids and Planet Wars in 2018 [20]. Given that the algorithm performed well when adapted to multiple specific games, including the challenging multi-agent game Hero Academy [13], it seemed natural to test its strength in the GVGAI framework. Several works have been published on such applications in recent years, ranging from analysis of the vanilla algorithm, to enhancements or hybrids meant to boost performance. These modifications, which will be described in more detail in the following subsections, inspired the work to be extended to General Game Playing domains with moderate success by Santos et al. [28].

5.1 Vanilla RHEA

RHEA in its vanilla form follows several simple steps, at every game tick, as depicted in Figure 6.

1. **Population initialisation**. All P individuals in the population are initialised as random action sequences of length L at the beginning of a game tick. For simplification and speed, individuals are represented by sequences of integers, where each gene can take a value between 0 and A, where A is the maximum number of available actions in the current game tick. Genes are always kept in this range through mutation, and are mapped back to game actions at the end of the evolution.
2. **Individual evaluation**. All individuals in the population are evaluated to assess their fitness. The actions in the sequence are executed, in turn, using a game model for simulations of possible future states given an action. The final state reached is evaluated with a heuristic H, and the value becomes the fitness of the individual.
3. **Order population** by fitness.
4. **Elitism**. The E best individuals are carried forward unchanged to the next generation.
5. **Individual selection**. Two parents are selected through tournament. [For $P > 1$]
6. **Offspring generation**. The parents are combined through uniform crossover to create a new individual (genes are randomly selected from the two parents to form a new individual). [For $P > 1$]
7. **Offspring mutation**. The offspring [or the only individual, if $P = 1$] is uniformly randomly mutated (genes are replaced with new random ones, with some probability M).
8. **Repeat steps 5-7** to create a new population of size P.
9. **Repeat steps 2-8** for N generations, or as long as the budget allows.
10. **Play first action of best individual** obtained at the end of the evolution process.

In GVGAI, vanilla RHEA uses a simple heuristic function H to evaluate game states, represented by the current game score (which it aims to maximise), to which a large integer is added if the game state is final and RHEA won, or a large integer is subtracted if the game state is final and RHEA lost.

Parameter analysis It is natural to notice several parameters in the steps described. Arguably the most important ones are P, the population size, and L, the length of the action sequences evolved. It is interesting to note that a popular approach in RHEA literature is to keep the population size to only 1 individual, turning the EA into a Random Mutation Hill Climber or (1+1) EA: in this scenario, the one individual is mutated at every generation and it is kept (and the first is discarded) if its fitness is better than the first, or discarded otherwise. This leads to a focused

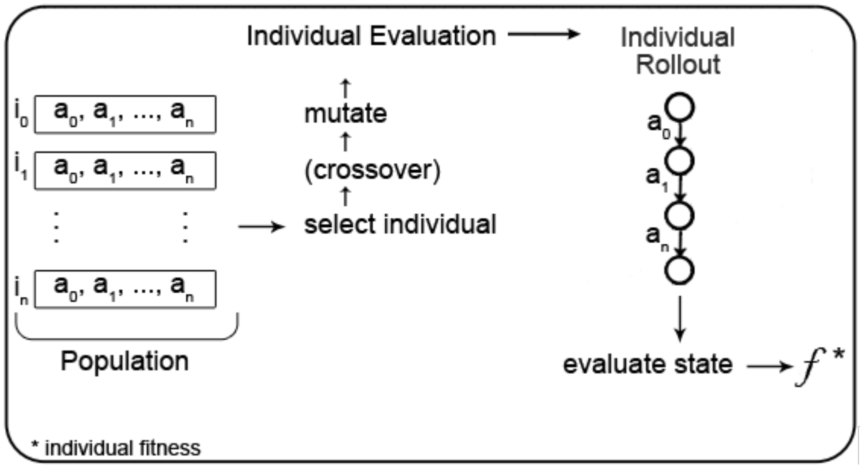

Fig. 6: Rolling Horizon Evolutionary Algorithm cycle.

fitness increase, with the risk of being stuck in local optima heavily relying on the chosen mutation operator.

Other parameters (such as elitism E or mutation rate M) can also influence the behaviour of the algorithm, but research looked in depth at how P and L affect performance in a set of 20 GVGAI games, when all other parameters are fixed [6]. The values of these parameters were explored within a fixed budget of 480 calls to the forward model of the games, the average achieved by MCTS in the larger GVGAI game corpus in the 40ms imposed by the framework for real-time decision making.

If both parameter values are increased so that only one generation can be created and evaluated, the algorithm becomes Random Search (RS) - that is, random action sequences are generated and the first action of the best sequence is played. No evolution takes place in this scenario.

The study carried out by Gaina et al. [6] found that, generally, the higher the L and P, the better - even in the extreme case where no evolution takes place. In the very limited budget, RHEA is unable to evolve better sequences than those randomly generated by RS, due to the big challenge of exploring quickly a large search space. In order for RHEA to be able to compete with RS and MCTS, it needs several enhancements that help it make better use of the budget and therefore search more efficiently (some will be detailed in the following section). The study does highlight that, given more budget, RHEA is able to outperform RS - therefore it is indeed simply a case of fast and efficient evolution being needed.

It is interesting, however, that high population sizes in RHEA lead, on average, to its ability to outperform MCTS nonetheless. Since most GVGAI competition entries (including past winners) are based on MCTS with several modifications, this is an indication that RHEA-based entries have the potential to be even better and definitely a viable alternative for general AI.

RHEA with very long sequences was also shown to perform better in [9], when budget is increased proportionally as well - although going beyond 150 appears to be detrimental especially in dense reward games.

Although on average more and longer individuals seem better, this does differ on a game by game basis: for example, in very dense reward games it may be more beneficial to have shorter individuals, since the reward landscape varies enough in the short-term for the agent to be able to tell which are good actions and which are not (although this could affect its performance in deceptive games in which short-term penalties could lead to greater long-term rewards). The differences in games seem important enough to warrant research in dynamic parameter adjustment for better flexibility, possibly in the form of a meta-heuristic that analyses the game being played (or even more granular, at game state level), and choosing the right configuration for the given scenario.

A first step in this direction was taken recently by Gaina et al. [9]. The approach taken in this study is at game state level, analysing the features observed by the agent about its decision making process in a given state in order to dynamically adjust the length of the individuals. In this case, only one feature was used, how flat the landscape fitness looked to the agent after all its simulations, and the length adjusted as per Algorithm 4, with frequency $\omega = 15$, a lower bound $SD_{-} = 0.05$ (which indicates the length will be increased if fitness landscape flatness value falls below this value) and an upper bound $SD_{+} = 0.4$ (which indicates the length will be decreased if fitness landscape flatness value rises above this value). The fitness landscape is represented by a collection of all fitness values observed during one game tick; the flatness value then becomes the standard deviation (δ) of all fitness values. This method therefore tries to adjust the length so that more individuals can be sampled if the landscape is varied, in order to gather enough statistics to make correct decisions; or so that long-term rewards can be found more easily with longer rollouts, if the fitness landscape is flat. It was shown to increase performance in sparse reward games for MCTS (while not affecting dense reward games), but it turned out to be detrimental for RHEA and halve its performance instead. This was thought to be due to the shift buffer enhancement (see next section), which is not compatible with dynamically adjusted sequence lengths.

Algorithm 4 Adjusting rollout length dynamically.

Input: t: current game tick
Input: Ld: the fitness landscape (all fitness values) observed in the previous game tick
Input: L: rollout length
Requires: ω: adjustment frequency
Requires: SD_-: lower f_{Ld} limit for L increase
Requires: SD_+: upper f_{Ld} limit for L decrease
Requires: M_D: rollout length modifier
Requires: MIN_L: minimum value for L
Requires: MAX_L: maximum value for L

```
1: function DYNLENGTH($Ld, t, L$) $t$ mod $\omega = 0$ $Ld = null$
2:     $f_{Ld} \leftarrow SD_-$                        ▷ $f_{Ld}$ is a measure of the fitness landscape flatness
3:     $f_{Ld} \leftarrow \delta(Ld)$                  ▷ get standard deviation $f_{Ld} < SD_-$
4:     $L \leftarrow L + M_D$ $f_{Ld} > SD_+$
5:     $L \leftarrow L - M_D$
6:     BOUND($L,\ MIN_L,\ MAX_L$)                      ▷ sequence length capped between min and max
7:
8: function BOUND($L,\ MIN_L,\ MAX_L$) $L < MIN_L$
9:     $L \leftarrow MIN_L$ $L > MAX_L$
10:    $L \leftarrow MAX_L$
11:    return $L$
```

5.2 RHEA Enhancements

As we've seen in the beginning of this section on Rolling Horizon Evolution, vanilla RHEA does not sample the search space efficiently enough to be able to find good solutions in the short budget allocated for real-time decision computation. This subsection therefore explores ways to improve upon the base algorithm by incorporating other techniques or even by combining RHEA with other algorithms for interesting and high-performing hybrids.

Population initialisation A first step towards vanilla RHEA improvement was taken in a 2017 study by Gaina et al. [7], which looked at the very first step of the algorithm described in Section 5.1: population initialisation. The theory behind it is that since RHEA cannot find a good enough solution quickly starting from random individuals, it would make sense that starting from an initially good solution could lead to better results.

There are many ways explored in literature to initialise evolutionary algorithms, although little for the specific application we have at hand here. Kazimipour et al. [14] present a nice review of such initialisation methods for evolutionary algorithms, looking at the randomness of the method, its generality or compositionality. Even though some of the methods described are hinted at working well within general settings as proposed in GVGAI, they are also noted to be computationally expensive and not directly applicable to real-time games. As we are interested in performance boosts within limited time budgets, these might not be the best option.

Research in the area is encouraged, however, by earlier research in the game Othello [15], which, even though still not real-time, showed significant improvement when the EA is initialised with an optimal solution determined by Temporal Difference Learning.

Following this line of research, [7] use two different algorithms to produce initial optimal solutions from which to start the evolutionary process: a One Step Look Ahead greedy algorithm (1SLA, which chooses simply the action which leads to the next best state in any given state) and MCTS. These algorithms are given a chunk of RHEA's thinking budget (half for MCTS, enough to produce one individual for 1SLA) to return one good solution, which becomes the first individual in the population. This first solution is then mutated to form the rest of the initial population, and evolution proceeds as before.

The effects of these seeding options were tested using different values for the P and L parameters, as these affect not only the number of generations RHEA can perform, but also how much budget the seeding algorithms have to be allocated in order to generate a full individual, as well as how much this initial individual is disturbed (most in high population sizes). Results showed that, generally, MCTS seeding leads to a significantly better performance, although it is also significantly worse in four of the games in which MCTS typically performs poorly (although in this form, MCTS-seeded RHEA still performs better than simply MCTS). This could indicate that RHEA is unable to change the initial solution provided enough to fully account for the weaknesses of MCTS, thus higher mutation rates or different operators could be needed in order to make the best of both algorithms.

The 1SLA seeding appeared to be detrimental in most cases, possibly due to the fact that RHEA was unable to escape the initial local optima provided through seeding. There were games, however, where even this method was better than vanilla RHEA, suggesting that dynamically changing the seeding method depending on game type (or, as seen before, more granular at state level) could significantly improve results in specific games, as well as on average.

Bandit-based mutation Since the mutation operator appeared to be one of the problems in the seeding-based methods described previously, it is natural to explore alternatives. One option that showed promise in other work [19] was a bandit-based mutation operator: this uses two levels of multi-armed bandit systems, one at individual level to choose which gene to mutate, and another at gene level to choose the new value given to the gene. Both of the systems employ the UCB equation (Equation 1) with a constant $C = \sqrt{2}$ in order to balance between exploration of potentially good values and exploitation of known good mutations. The Q(s,a) values are updated based on the new individual fitness, including the option to revert the mutation if this proved to be detrimental to the action sequence, aiming to always improve individuals.

Although this worked in previous applications of the method, it performed very poorly overall in the tests performed by Gaina et al. [8] on GVGAI games, in most cases being worse than vanilla RHEA, even when increasing the population size and individual length to get the best performance out of this hybrid. The bad performance is most likely due to the fact that changing one gene in the middle of the action sequence affects the meaning of the following actions as well - and it could be that one of the other actions were the ones producing the change in fitness wrongly attributed to the mutated gene instead - therefore a better calculation of the value of the mutation could potentially improve the performance of this enhancement. This focused mutation for improvement also has the potential of getting the algorithm stuck in local optima.

Statistical tree Similar to the work in [25] which showed promise, it's possible to keep more statistics throughout the evolutionary process. Adopting the way MCTS computes statistics about the actions it explores, but without relying the search on these statistics (so the search would still be performed by the regular evolutionary algorithm previously described), is an interesting way of deciding which action to finally play. This final decision would be based in this case on the action at the root of the tree with the highest UCB value, instead of the first in the best plan evolved. Figure 7 shows how the actions would be stored in the statistical tree after every individual rollout, using the individual fitness to backup the values throughout all the actions.

This enhancement did work fairly well in the tests in [8], performing the best (and better than vanilla RHEA) in low configurations for the parameters P and L, thus when enough individuals were evaluated to build significant statistics in the tree. In high configurations the algorithm was unable to gather enough statistics, suggesting it is better to play the first action of the best overall plan evolved instead. This could suggest further applications in dynamic parameter configuration work: switching this enhancement on when parameter values are low enough could lead to further boost in performance (e.g. compared to the work described in [9].

Shift buffer The next enhancement we're going to discuss is partly related to the first step of the algorithm as well (population initialisation, that is). We say, partly,

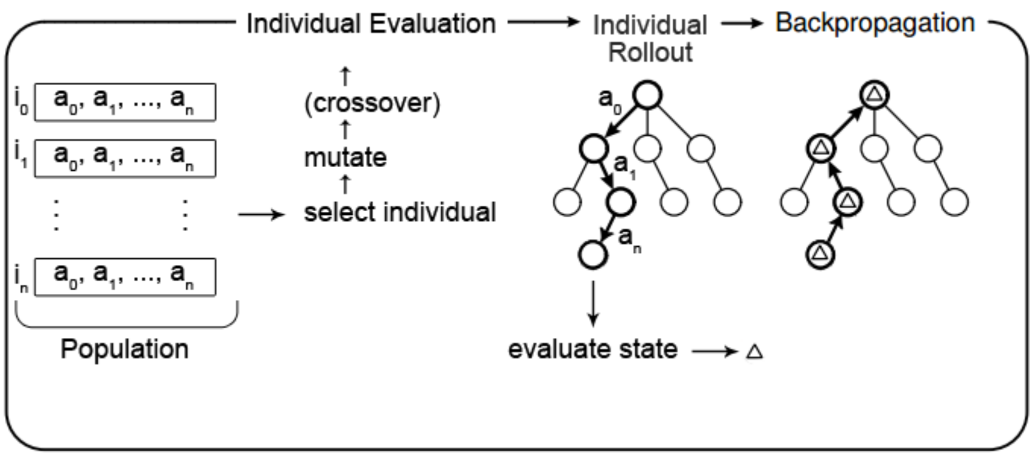

Fig. 7: RHEA statistical tree steps.

because it does not actually form a new initial population. This is instead a method of keeping the evolved population of individuals between game ticks, instead of starting from new random points each time.

A shift buffer refers to the technique through which all individuals in the final population in one game tick become the individuals in the first population in the next game tick. However, because the first action in the sequences evolved was played, we need to shift the search horizon to start from the second action in each individual and bring them up to date to the current game time. Therefore, the first action in all individuals is removed, while a new random action is added at the end, in order to keep the same length of individuals. In order to make sure all actions are still legal in case a change in the action space occurred during game ticks, any illegal actions are replaced with new random actions as well.

This enhancement showed promise when applied to MCTS within the context of the Physical Travelling Salesman Problem [25] and, even though it does not appear to work as well for MCTS in GVGAI, it does help RHEA make better use of limited thinking budgets by reusing already evolved populations and getting to improve the plans further instead of discarding all of its computed information.

In the study by Gaina et al. [8], the addition of this enhancement led to a high win rate increase (as well as significantly higher scores) over vanilla RHEA in all configurations. The shift buffer is not the best combination with bandit mutation, possibly due on one hand to the general poor performance of bandit variations, but also due to the old statistics used by the bandits to make their recommendations. The success of this variant encouraged its use in several other works, such as that by Santos et al. [28] who use a Rolling Horizon Evolutionary Algorithm with a shift buffer for General Game Playing, with good results.

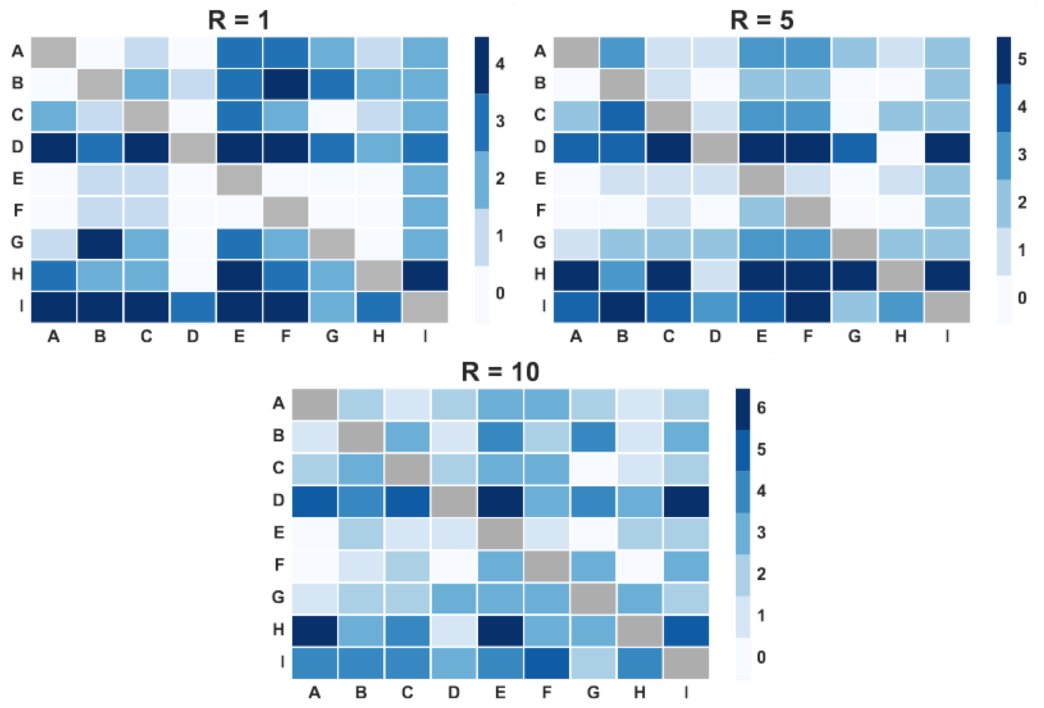

Fig. 8: Win percentage for configuration 10-14. The color bar denotes in how many unique games row was significantly better than column. Legend: A = Vanilla, B = EA-roll, C = EA-shift, D = EA-shift-roll, E = EA-tree, F = EA-tree-roll, G = EA-tree-shift, H = EA-tree-shift-roll, I = MCTS

Monte Carlo rollout evaluation One last enhancement that's been explored in research [12] involves step 2 of the vanilla RHEA algorithm (individual evaluation) and is inspired by Monte Carlo rollouts as in MCTS. The idea here is that after finishing the usual advancing through the actions in the sequence, we add further horizon to the search with a random rollout of length r_L, possibly sampled multiple times, r_N, so that the results are more significant. In this case, the fitness of the individual is instead given by the average values of states reached after r_L more actions executed at the end of the action sequence, as in Equation 10.

$$f = \frac{\sum_{n=1}^{r_N} V(s_n)}{r_N} \tag{10}$$

As opposed to the case where the sequence length L is increased directly, this variant offers the possibility of exploring more varied (and not fixed) action sequences, which allows it to find interesting variations in the search space it might not otherwise. In the study by Gaina et al. [8], r_L is given a value as half the length of the individual ($L/2$), while r_N is tested for different values: 1, 5 and 10. This variant was tested individually against vanilla RHEA, but also in combination with previously described enhancements (with the exception of the bandit-based mutation, which was considered to be performing too poorly to consider for this last experiment). An overview of these results can be observed in Figure 8; MCTS is also included in the figure for comparison. A further summary of the best variants obtained in all configuration of parameters P and L can also be observed in Table 5.

Overall, Monte Carlo rollouts at the end of individual evaluation appeared to offer a nice boost in performance, especially when combined with a shift buffer, significantly outperforming vanilla RHEA (likewise, the shift buffer enhancement is even better if in combination with MC rollouts). As may be expected, however, MC rollouts work best if P and L are lower, due to the limited budget in higher configurations that would now have to be split between action sequences and MC rollouts as well. These computations become fairly expensive as individual length grows, while still yielding good results.

Regarding r_N values, it appeared that 5 was best in many cases. A highlight of this is given by the variant combining the shift buffer and MC rollouts, which matches the performance of MCTS when $r_N = 5$.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.
2. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4:1, pp. 1–43, 2012.
3. G. M. J.-B. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-Carlo Tree Search: A New Framework for Game AI," in *Proceedings of the Artificial Intelligence for Interactive Digital Entertainment Conference*, 2006, pp. 216–217.
4. R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Proceedings of the 5th International Conference on Computer Games.* Springer-Verlag, 2006, pp. 72–83.

5. H. Finnsson, "Generalized Monte-Carlo Tree Search Extensions for General Game Playing." in *AAAI*, 2012.
6. R. D. Gaina, J. Liu, S. M. Lucas, and D. Pérez-Liébana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *European Conference on the Applications of Evolutionary Computation.* Springer, 2017, pp. 418–434.
7. R. D. Gaina, S. M. Lucas, and D. Pérez-Liébana, "Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing," in *Conference on Evolutionary Computation.* IEEE, 2017.
8. ——, "Rolling Horizon Evolution Enhancements in General Video Game Playing," in *2017 IEEE Conference on Computational Intelligence and Games (CIG).* IEEE, 2017.
9. R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Tackling Sparse Rewards in Real-Time Games with Statistical Forward Planning Methods," in *AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.
10. S. Gelly and D. Silver, "Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.
11. S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with Patterns in Monte-Carlo Go," Inst. Nat. Rech. Inform. Auto. (INRIA), Paris, Tech. Rep., 2006.
12. H. Horn, V. Volz, D. Pérez-Liébana, and M. Preuss, "MCTS/EA Hybrid GVGAI Players and Game Difficulty Estimation," in *Conference on Computational Intelligence in Games (CIG).* IEEE, 2016, pp. 1–8.
13. N. Justesen, T. Mahlmann, and J. Togelius, "Online evolution for multi-action adversarial games," in *European Conference on the Applications of Evolutionary Computation.* Springer, 2016, pp. 590–603.
14. B. Kazimipour, X. Li, and A. K. Qin, "A review of population initialization techniques for evolutionary algorithms," in *Evolutionary Computation (CEC), 2014 IEEE Congress on.* IEEE, 2014, pp. 2585–2592.
15. K.-J. Kim, H. Choi, and S.-B. Cho, "Hybrid of evolution and reinforcement learning for othello players," in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on.* IEEE, 2007, pp. 203–209.
16. L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," *Machine Learning: ECML 2006*, vol. 4212, pp. 282–293, 2006.
17. 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.
18. J. Liu, D. Pérez-Liébana, and S. M. Lucas, "Rolling horizon coevolutionary planning for two-player video games," in *Computer Science and Electronic Engineering (CEEC), 2016 8th.* IEEE, 2016, pp. 174–179.
19. ——, "Bandit-based random mutation hill-climbing," in *Evolutionary Computation (CEC), 2017 IEEE Congress on.* IEEE, 2017, pp. 2145–2151.
20. S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," *arXiv preprint arXiv:1802.05991*, 2018.
21. S. M. Lucas, S. Samothrakis, and D. Perez, "Fast Evolutionary Adaptation for Monte Carlo Tree Search," in *Proceedings of EvoGames*, 2014, p. to appear.
22. R. B. Myerson, *Game Theory : Analysis of Conflict.* Cambridge (Mass.), London: Harvard university press, 1997, 1991.
23. D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation.* ACM, 2013, pp. 351–358.
24. D. Perez, S. Samothrakis, J. Togelius, T. Schaul, and S. Lucas, "The General Video Game AI Competition," 2014, www.gvgai.net.
25. D. Perez Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. Lucas, "Open Loop Search for General Video Game Playing," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation.* ACM, 2015, pp. 337–344.

26. D. Perez-Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. M. Lucas, "Open Loop Search for General Video Game Playing," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*. Association for Computing Machinery (ACM), 2015, pp. 337–344.
27. D. Perez-Liebana, S. Mostaghim, and S. M. Lucas, "Multi-objective Tree Search Approaches for General Video Game Playing," in *Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 624–631.
28. B. Santos, H. Bernardino, and E. Hauck, "An improved rolling horizon evolution algorithm with shift buffer for general game playing," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2018, pp. 31–316.
29. 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.

Game	Algorithm	Victories (%)	τ (Vict.)	Scores	τ (Score)
G1	A: Sample MCTS	78.00 (1.85)	B , C	54.60 (0.39)	B
	B: Weighted Sum MCTS	64.00 (2.15)	Ø	53.91 (0.40)	Ø
	C: Mixed Strategy MCTS	67.00 (2.10)	Ø	54.44 (0.40)	Ø
	D: MO-MCTS	99.80 (0.20)	A , B , C	60.17 (0.50)	A , B , C
G2	A: Sample MCTS	0.25 (0.25)	Ø	6.83 (0.23)	Ø
	B: Weighted Sum MCTS	0.00 (0.00)	Ø	7.26 (0.26)	Ø
	C: Mixed Strategy MCTS	0.32 (0.32)	Ø	6.76 (0.25)	Ø
	D: MO-MCTS	0.40 (0.28)	Ø	7.53 (0.21)	Ø
G3	A: Sample MCTS	98.20 (0.59)	B , D	28.88 (0.62)	Ø
	B: Weighted Sum MCTS	95.00 (0.97)	D	28.84 (0.64)	Ø
	C: Mixed Strategy MCTS	98.00 (0.63)	B , D	30.02 (0.66)	Ø
	D: MO-MCTS	91.40 (1.25)	Ø	32.11 (0.72)	A , B , C
G4	A: Sample MCTS	3.20 (0.79)	B , C	2.47 (0.09)	B , C
	B: Weighted Sum MCTS	0.00 (0.00)	Ø	1.07 (0.06)	Ø
	C: Mixed Strategy MCTS	0.00 (0.00)	Ø	1.37 (0.06)	B
	D: MO-MCTS	1.80 (0.59)	B , C	2.49 (0.09)	B , C
G5	A: Sample MCTS	13.60 (1.53)	Ø	-1.36 (0.05)	Ø
	B: Weighted Sum MCTS	11.28 (1.47)	Ø	-1.41 (0.05)	Ø
	C: Mixed Strategy MCTS	11.20 (1.41)	Ø	-1.39 (0.05)	Ø
	D: MO-MCTS	86.40 (1.53)	A , B , C	0.75 (0.03)	A , B , C
G6	A: Sample MCTS	36.00 (2.15)	Ø	0.68 (0.17)	Ø
	B: Weighted Sum MCTS	40.80 (2.20)	Ø	0.86 (0.19)	Ø
	C: Mixed Strategy MCTS	38.00 (2.17)	Ø	0.67 (0.17)	Ø
	D: MO-MCTS	55.60 (2.22)	A , B , C	3.47 (0.21)	A , B , C
G7	A: Sample MCTS	15.80 (1.63)	Ø	0.16 (0.02)	Ø
	B: Weighted Sum MCTS	16.40 (1.66)	Ø	0.16 (0.02)	Ø
	C: Mixed Strategy MCTS	14.20 (1.56)	Ø	0.14 (0.02)	Ø
	D: MO-MCTS	41.60 (2.20)	A , B , C	0.42 (0.02)	A , B , C
G8	**A: Sample MCTS**	24.60 (1.93)	B , C , D	1.28 (0.04)	B , C , D
	B: Weighted Sum MCTS	16.80 (1.67)	Ø	0.81 (0.04)	Ø
	C: Mixed Strategy MCTS	15.60 (1.62)	Ø	0.96 (0.03)	B
	D: MO-MCTS	15.60 (1.62)	Ø	1.09 (0.04)	B , C
G9	A: Sample MCTS	41.00 (2.20)	D	33.72 (1.31)	B
	B: Weighted Sum MCTS	38.00 (2.17)	D	29.92 (1.14)	Ø
	C: Mixed Strategy MCTS	42.20 (2.21)	D	32.54 (1.23)	Ø
	D: MO-MCTS	22.20 (1.86)	Ø	37.68 (1.21)	A , B , C
G10	A: Sample MCTS	5.40 (1.01)	Ø	2.26 (0.12)	Ø
	B: Weighted Sum MCTS	4.20 (0.90)	Ø	2.41 (0.13)	Ø
	C: Mixed Strategy MCTS	7.20 (1.16)	B	2.17 (0.12)	Ø
	D: MO-MCTS	9.00 (1.28)	A , B	3.69 (0.14)	A , B , C

Table 4: Victory rate and score average (standard error). τ columns indicate significant dominance (Wilcoxon signed-rank test, p-value < 0.05) and bold font indicates dominance over the other three in victories or score. Games listed in the order used in Figure 3.

Config.	**R**	**Best By F1 Points**		**Best By Win Rate**	
		Algorithm	**Avg. Wins**	**Algorithm**	**Avg. Wins**
1-6	1	**EA-shift-roll**	38.35 (2.31)	**EA-tree-shift-roll**	38.60 (2.55)
	5	**EA-shift-roll**	40.10 (2.51)	**EA-shift-roll**	40.10 (2.51)
	10	**EA-shift-roll**	39.35 (2.64)	**EA-shift-roll**	39.35 (2.64)
2-8	1	**EA-shift-roll**	40.35 (2.63)	**EA-shift-roll**	40.35 (2.63)
	5	**EA-shift-roll**	40.75 (2.46)	**EA-shift-roll**	40.75 (2.46)
	10	**EA-shift-roll**	40.20 (2.30)	**EA-shift-roll**	40.20 (2.30)
5-10	1	**EA-shift-roll**	43.20 (2.43)	**EA-shift-roll**	43.20 (2.43)
	5	**EA-shift**	40.05 (2.50)	**EA-shift-roll**	41.85 (2.42)
	10	**EA-shift**	40.05 (2.50)	**EA-shift**	40.05 (2.50)
10-14	1	**EA-shift**	39.75 (2.54)	**EA-shift-roll**	42.80 (2.44)
	5	**EA-shift-roll**	42.05 (2.48)	**EA-tree-shift-roll**	42.70 (2.41)
	10	**EA-shift-roll**	42.35 (2.53)	**EA-shift-roll**	42.35 (2.53)

Table 5: The best algorithms (by Formula-1 points and win rate) in all configurations and rollout repetitions (R), as compared against the other variants in the same configuration and the same R value (includes variants without rollouts).