Open Loop Search for General Video Game Playing

Diego Perez[*]
diego.perez@ovgu.de

Jens Dieskau
jens.dieskau@st.ovgu.de

Martin Hünermund
martin.huenermund@gmail.com

Sanaz Mostaghim
sanaz.mostaghim@ovgu.de

Simon M. Lucas
sml@essex.ac.uk

ABSTRACT

General Video Game Playing is a sub-field of Game Artificial Intelligence, where the goal is to find algorithms capable of playing many different real-time games, some of them unknown a priori. In this scenario, the presence of domain knowledge must be severely limited, or the algorithm will overfit to the training games and perform poorly on the unknown games of the test set. Research in this area has been of special interest in the last years, with emerging contests like the General Video Game AI (GVG-AI) Competition. This paper introduces three different open loop techniques for dealing with this problem. First, a simple directed depth first search algorithm is employed as a baseline. Then, a tree search algorithm with a multi-armed bandit based tree policy is presented, followed by a Rolling Horizon Evolutionary Algorithm (RHEA) approach. In order to test these techniques, the games from the GVG-AI Competition framework are used as a benchmark, evaluation on a training set of 29 games, and submitting to the 10 unknown games at the competition website. Results show how the general game-independent heuristic proposed works well across all algorithms and games, and how the RHEA becomes the best evolutionary technique in the rankings of the test set.

Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence, Learning

Keywords

Games; Decision Making; Genetic Algorithms

1. INTRODUCTION

Research in Game AI traditionally presents algorithms that play a particular game, counting on heuristics tailored to them. These heuristics are usually designed manually by some expert in these specific games, and are used to guide the search to states with high chances of victory.

General Game Playing (GGP) and General Video Game Playing (GVGP), on the other hand, drive research to algorithms that play a set of significantly different games, even some of them unknown a priori. Therefore, the presence of specific domain knowledge is limited or almost non-existent. The main challenge of GGP and GVGP is that the agent needs to be general enough to learn the structure of any game, guide the search to relevant states of the search space, and be able to adapt to a wide variety of situations in the absence of game-dependent heuristics. The idea of having general agents that are able to deal with a large variety of situations is one of the most important goals of AI [19].

One of the first attempts to develop and establish a GGP framework was carried out by the Stanford Logic Group of Stanford University, when they organized the first Association for the Advancement of Artificial Intelligence (AAAI) GGP competition in 2005 [6]. GGP focuses mainly on 2-player board games, where each player must provide an action after 1s of “thinking time”. In contrast, GVGP proposes playing real-time single-player games, where the time budget for a move is measured in milliseconds. Recently, a GVGP framework and competition has been carried out by Perez et al. [17]. Interestingly, both GGP and GVGP competitions have shown the proficiency of Monte Carlo Tree Search (MCTS) methods, the winner of several editions of these contests. For instance, CADIA-Player (by Hilmar Finnsson [5]), was the first MCTS based approach to be declared winner of the 2007 AAAI GGP competition, followed by Ary, another MCTS approach developed by J. Méhat and T. Cazenave [12] that won the 2009 and 2010 editions. In the case of GVG-AI, the winner of the only edition of the competition (2014) was Adrien Couëtoux with an Open Loop MCTS approach (OLETS) [17].

However, to the knowledge of the authors of this paper, no research has been done on Rolling Horizon Evolutionary Algorithms (RHEA) for GVGP. This paper addresses this, proposing a combination of an evolutionary algorithm with a game tree search, and compares it with other tree search methods. Additionally, this paper proposes a game-independent GVGP heuristic that focuses on maximizing the exploration of the level played by the agent. Last but not least, this paper analyzes the hazards posed by stochastic real-time games to action-decision making, and the advantages of using an open loop approach to deal with these. Most of the agents submitted to the GVG-AI Competition were based on closed loop techniques [17].

This paper is structured as follows. Section 2 presents the benchmark used for this research. Then, Section 3 explains the concepts behind the agents proposed in this paper, such as game tree search, open loop and RHEA. Section 4 describes the algorithms used for

[*] The first four authors are with the Faculty of Computer Science at the Otto von Guericke University, Magdeburg, Germany. Simon M. Lucas is with the Computer Science and Electronic Engineering School at the University of Essex, Colchester, UK.

GECCO ’15, July 11–15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3472-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739480.2754811

the experimental study, detailed in Section 5. Finally, Section 6 draws some final conclusions and lines for future work.

2. THE GVG-AI FRAMEWORK

The General Video Game Playing competition and framework (GVG-AI) is inspired by the Video Game Description Language (VGDL), a framework designed by Ebner et al. [4] and developed by Tom Schaul [22] for GVGP in Python (*py-vgdl*). In VGDL, *objects* interact in a two-dimensional rectangular space, with associated coordinates, and have the ability to interact (collide) with each other. VGDL defines an ontology that allows the generation of real-time games and levels with simple text files.

The GVG-AI framework is a Java port of py-vgdl, re-engineered for the GVG-AI Competition by Perez et al. [17], which exposes an interface that allows the implementation of controllers that must determine the actions of the player (also referred to as the *avatar*). The framework provides a *forward model* and information regarding the state of the game to the controllers. The VGDL definition of the game is, however, not provided, so it is the responsibility of the agent to determine the nature of the game and the actions needed to achieve victory.

A controller in the GVG-AI framework can spend 1s of initialization time before a game starts, and 40ms for each game step, where a move to execute in the game must be returned. During each one of these calls, the agent receives information about the current state of the game, such as the current time step, game score, current victory condition (game *won*, *lost* or *ongoing*) and the list of available actions in the game (variable from game to game, these can be moving *left*, *right*, *up* or *down*, and a generic *use* action). Information about the observations in the level (other sprites in the game) and history of avatar events (collisions between the avatar and other sprites) is also provided. The GVG-AI framework includes several sample controllers. The most relevant to our research are *SampleGA* (a simple RHEA), and *SampleMCTS* and *SampleOLMCTS* (closed and open loop MCTS approaches, respectively).

The GVG-AI framework featured in the first edition of the GVG-AI Competition in 2014, held in the IEEE Computational Intelligence in Games Conference in Dortmund (Germany), and the second edition will be held in the Genetic and Evolutionary Computation Conference (GECCO) 2015 in Madrid (Spain). At the time of the writing of this paper, 29 games of the framework have been made public (20 used in the competition, plus 9 created after the contest), which form the training game set used for the experiments described in this paper. 10 more games, unknown to all participants, are held in the server. Contestants can submit their controllers to this server in order to play these secret games. This set of 10 games is used as a test set for this research. All games vary in many different aspects, such as winning conditions, number and types of non-player characters, scoring mechanics and even in the available actions for the agent. Examples of known games are *Sokoban*, *BoulderDash*, *Zelda*, *Frogger*, *Digdug* and *Pacman*. The full list of games, sample controllers, results and rules of the competition can be found on the website[1] and in [17].

The results of the 2014 GVG-AI Competition show that this problem is far from trivial. The winner of the contest achieves a rate of victories, in the test set, of 51.2%, and only 2 more entries were able to obtain a rate higher than 30%. The way that entries are ranked, however, is not by the percentage of victories across this set of 10 games. Instead, controllers are sorted (for each game) by the highest number of games won, the highest total sum of game scores (as a tie-breaker), and finally the minimum time spent in total (as a second tie-breaker). Points are then awarded to them according to this order, following a Formula-1 score system (25, 18, 15, 12, 10, 8, 6, 4, 2 and 1 points; entries beyond the 10^{th} position obtain 0 points). The winner is determined by the sum of points across all games, using the number of first, second, etc. positions to break possible ties.

3. BACKGROUND

This section introduces the main techniques behind the algorithms presented in this paper, such as game tree search, open loop approaches and RHEA.

3.1 Game Tree Search

Tree Search (TS) is one of the most employed types of techniques for action-decision making in games. Tree search techniques are iterative approaches originally employed for 2-player board games such as Chess, Checkers or Go. Research in this type of algorithms has advanced the state of the art from initial approaches such as Minimax or $\alpha - \beta$ search [20] to Monte Carlo Tree Search (MCTS) [3, 2]. This and other tree search algorithms are currently applied to a wide range of games, including games of uncertain information, general game playing, single-player games and real-time games.

A tree search algorithm builds a *game tree* with game states as nodes and actions as edges, with the current game state as the root of the tree. One of the most important components needed by a tree search algorithm is a *generative* or *forward model*, in order to be able to simulate actions before making a final decision about the move to make. In particular, a forward model is able to produce a state s' from a state s when an action a is taken ($a \in A(s)$ is one of the possible actions from the state s).

When the tree reaches a terminal state (where the game is over), it is possible to determine if the game was won or lost. However, in some cases, terminal states are so far ahead in the future that they are not reachable by simulations (this happens, for instance, in real-time games, a subject of study in this paper). In this situation, the algorithm defines a Simulation Depth D, that limits the number of moves that are performed from the current state, and a *heuristic* function, which evaluates a non-terminal state.

In either case, the evaluation of a game state allows statistics to be stored in the nodes of the tree regarding how good or bad certain actions are, and therefore bias the action selection process and tree growth towards the most promising parts of the search space. Examples of statistics that are usually stored are: how often each move is played from a given state ($N(s, a)$), how many times any move is played from there ($N(s)$) and the empirical average of the rewards obtained when choosing action a at state s ($Q(s, a)$).

This information allows the agent to make decisions at two levels. First, for each iteration of the algorithm, decisions must be made while navigating the tree (a procedure known as *Tree Policy*, that determines actions to take from a given state). Difference policies are used in the literature, such as ϵ-greedy (with $1 - \epsilon$ probability, the best action from a state is chosen; otherwise, another action is picked uniformly at random) or Upper Confidence Bound 1 (UCB1) , introduced by Kocsis [9] and shown in Equation 1:

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

This equation finds an action a that maximizes the UCB1 equation, designed to provide a balance between exploration of the different actions available, and exploitation of the most promising action found so far. $Q(s, a)$ is the exploitation term, while the second

[1] http://www.gvgai.net

term (weighted by a constant C) is the exploration term. If the 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 value of C is application dependant, and it may vary from game to game. Additionally, when $N(s,a) = 0$ for any of the possible actions from s, the action a must be chosen before UCB1 can be applied. This tree selection policy has become very popular because of the success of MCTS, concretely in the Upper Confidence Bounds for Trees version of the algorithm (UCT, [9]).

Secondly, a decision has to be made about the action to take when the iterations have finished (a limit imposed, for instance, by a time budget in real-time games). This policy, known as *Recommendation Policy*, can be implemented in different ways, such as selecting the action that leads to the highest reward found, the one with the highest average reward, the one with the maximum number of visits, or simply applying the UCB1 Equation 1.

3.2 Open versus Closed Loop Tree Search

In deterministic scenarios, a game tree can contain future states in its nodes that mimic perfectly the states that will be found when performing an action or sequence of actions (*Closed Loop* tree search). Additionally, once a new node is added to the tree, there is no need to generate such state again, and the algorithm can traverse the tree during the tree policy phase without spending time in using the forward model again from the previous pair (s,a).

However, this aspect changes when considering stochastic scenarios, as this poses the problem of not knowing how believable the states reached after a simulated move are. In the general case, given a state s and a set of actions $A(s)$, if the action $a \in A(s)$ is selected, one may assume that the state s' obtained using the forward model is drawn from an unknown probability distribution. Consequently, storing a state in a node could be detrimental: the state generated when the node was added to the tree might not be representative of the most probable s' found after applying a from s. Furthermore, even if it were, the search performed from that node in the tree would be biased by one instance of all possible states s'.

Ideally, in order to deal with this scenario, the algorithm should sample each one of the available actions a *sufficient* number of times, and create as many children as future states found. In practice, however, the combination of nodes that the tree would contain by applying this procedure grows exponentially. Additionally, the limited time budget available in real time games draws this solution infeasible.

An approximate solution to this problem is to use an *Open Loop* approach. The idea is that the nodes do not store the states, but only the statistics. Obviously, this forces the use of the forward model every time the algorithm chooses an action during the tree policy. However, now the statistics gathered are representative of the set of states found after applying an action a from s. In other words, the tree stores statistics derived from executing a sequence of actions from a given state (open loop case), and not from a sequence of pairs (s,a) (closed loop). More information about open versus closed loop control can be found in [23].

Another important detail about tree search is the way the information is kept. Typically, an algorithm runs during the allowed time budget and, at the end, selects an action to apply in the real game guided by the recommendation policy. This process is repeated during several game steps until the game ends. All controllers employed in this research keep this information from one game cycle to the next, as the data from the node that follows the chosen action can be reused. This node becomes the new root, and its siblings are discarded, as they can no longer be reached. Figure 1 shows an explanatory diagram of this procedure.

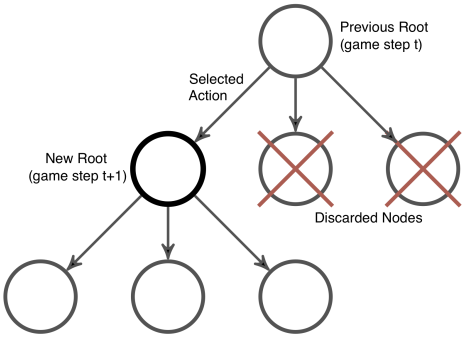


Figure 1: Changing the root node from a previous game step.

3.3 Rolling Horizon Evolutionary Algorithms

Rolling Horizon Evolutionary Algorithms (RHEA) are a suitable alternative to Tree Search for action-decision making in real-time games. Traditionally, Evolutionary Algorithms (EA) are used in conjunction with a simulator to train a controller offline, and then use the already evolved controller to play the game. This is a technique not only used in games [11], but also in other problems [8] or in the domain of Artificial Life [1].

RHEA approaches, however, employ evolution in a similar way to how it is done in tree search, using a forward model to evaluate a sequence of actions. The agent evolves a plan (a sequence of actions) during some predefined time budget, and then selects the first action of the best individual as the move to play in the real game. This type of approach is called “Rolling Horizon”, as planning is performed at every time step, up to a determined point in the future, as far as the length of the individual (or plan).

It is worthwhile mentioning that this approach, by definition, is an open loop approach, as the states found during the search are not stored or reused in any way. The evaluation of each individual consists of applying the sequence of actions, and assigning as fitness the value given by the evaluation of the state reached at the end. It is possible to find usages of RHEA in the literature in deterministic games, as in S. Samothrakis and S. Lucas [21] in the Mountain Car problem, or Perez et al. [16] in the Physical Travelling Salesman Problem.

A possible usage of RHEA in stochastic scenarios could be to evaluate each individual several times, and assign as fitness the average of these evaluations. However, the same problem seen in tree search affects RHEA in real-time games: the limited time budget does not permit running too many evaluations. This paper proposes the combination of RHEA and Tree Search, while keeping the benefits of open loop search, in one of the algorithms presented (see Section 4.4).

4. CONTROLLERS

This section explains the different controllers used for the experiments described later in Section 5. All these algorithms build a game tree without storing the states in the nodes (i.e., they implement open loop search, as explained in Section 3.2) and reusing the subtree in the next step. Additionally, all algorithms use the same heuristic to evaluate states (see Section 4.1) and the same recommendation policy (described in Section 4.2).

Algorithm 1 Pseudocode of the heuristic

```
 1: procedure HEURISTIC($state$, $avatar$, $pheromones$)
 2:     $p \leftarrow avatar.position$
 3:     $reward \leftarrow 0$
 4:     if HasPlayerWon($state$) then
 5:         $reward \leftarrow HIGH\_VALUE$
 6:     if HasPlayerLost($state$) then
 7:         $reward \leftarrow -HIGH\_VALUE$
 8:     $reward \leftarrow reward+$ GetGameScore($state$)
 9:     if OppositeActions($avatar$) then
10:         $reward \leftarrow reward - PEN_{OA}$
11:     if BlockedMovement($avatar$) then
12:         $reward \leftarrow reward - PEN_{BM}$
13:     $reward \leftarrow reward - pheromones[p.x][p.y]$
14:     return $reward$
```

4.1 Heuristic

All controllers presented in this study follow the same heuristic, whose objective is to guide the search and evaluate states found during the simulations. Note that this a domain independent heuristic, as it has been designed without any domain knowledge from specific games. Algorithm 1 shows how the different components of the heuristic are combined together.

The heuristic is a combination of several factors. First of all, the end condition of the game is checked (lines 4 to 7). When a game ends, the player may have won or lost the game. If any of these two scenarios is met, a very high or low reward is assigned to the score respectively.

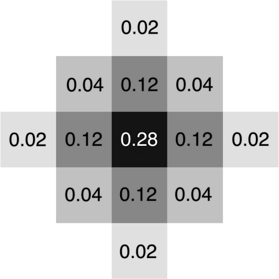
Then, the reward is modified by adding the current score of the game (line 8) and subtracting two penalties (lines 9 to 12):

- *Opposite Actions:* None of the algorithms tested impose any restriction about the movements available, other than the actions allowed by each game on each state. In some cases, it is possible to perform two consecutive contrary movements, such as moving *Right* after *Left* or vice-versa. With the exception of some rare cases where this is a good thing to do, this redundancy is unnecessary. Thus, a penalty PEN_{OA} is subtracted from the reward when this happens. The value of PEN_{OA} is set to 0.1, determined after a trial and error process.

- *Blocked Movement:* It is possible that one of the available movement actions in a state does not change the avatar's position in a determined situation (i.e. moving against a static wall). When this happens in the state reached, the reward receives a penalty PEN_{BM} set to -100, a value determined by trial and error.

Another possibility to these penalties would have been to deny some actions from being chosen, similar to pruning branches in tree search. However, penalties permit to achieve a similar effect with a lower computational cost, which is a valuable resource in real-time games. Additionally, this sort of *soft pruning* does not completely avoid an action from being selected, allowing some exploration through that node anyway. Also, there are some minor cases where taking these actions may be beneficial, such as dodging an bullet or not moving at all.

The last component of the heuristic is a nature-inspired technique based on *pheromone* trails, that works as a *potential field* [24] for the avatar. In the implementation presented in this paper, the avatar secretes repelling pheromones. These pheromones spread into the neighbouring area, defined as the grid cells above, below, on the right and on the left of a given position. The amount of pheromone contained in a given cell is a value between 0 and 1, and it decays with time. Figure 2 shows an example of a pheromone diffusion in one game step, where the avatar is located at the center.



		0.02		
	0.04	0.12	0.04	
0.02	0.12	0.28	0.12	0.02
	0.04	0.12	0.04	
		0.02		

Figure 2: Pheromone diffusion.

The amount of pheromone on each cell is updated at every state as shown in Equation 2:

$$\begin{aligned} pheromones_{i,j} &= \rho_{df} \times \rho_{\phi} + \\ &= (1-\rho_{df}) \times \rho_{dc} \times pheromones_{i,j} \end{aligned} \qquad (2)$$

where i and j are coordinates in the level board, and ρ_{ϕ} is the ratio of pheromone trail in the neighbouring cells divided by the number of neighbouring cells (note that the edges of the level might limit the amount of neighbours). $\rho_{df} \in (0, 1)$ defines the value of diffusion of the pheromone in the current game step, and $\rho_{dc} \in (0, 1)$ indicates the decay of the value at each game step. After some experimental testing, these values have been set to $\rho_{df} = 0.4$ and $\rho_{dc} = 0.99$, providing a balance between global and local exploration (higher and lower values, respectively).

The pheromone diffusion algorithm creates a high concentration of pheromones in the close proximity of the current and recent positions of the avatar. As defined in the heuristic (see line 13), the value of the pheromone is subtracted from the reward. Hence, the heuristic will give less value to those states with positions where the avatar is, or has been in recent time steps, aiming to increase the exploration of the level. Interestingly, this exploration technique is similar to the one developed independently (and almost simultaneously) by Nielsen et al. [13].

Summarizing, the heuristic employed to evaluate states rewards high game score or victory, plus favouring those actions that allow a larger exploration of the level the avatar is playing.

4.2 Recommendation Policy

The recommendation policy of the algorithms described in this paper selects the action a, so it maximizes the value $R(s, a)$ calculated for each one of the children of the current state s. Equation 3 shows how the value $R(s, a)$ is obtained[2]:

$$R(s,a) = w_r \times R_{max}(s,a) + (1 - w_r) \times Q(s,a) \qquad (3)$$

$R(s, a)$ is a weighted sum between $R_{max}(s, a)$ and the average of rewards obtained from state s after applying a (this is, $Q(s, a)$). $R_{max}(s, a)$ is described in Equation 4 as the maximum value of $R(s', a')$, where s' is the state reached from s after applying a, and a' is each one of the actions available from s'.

[2] In order to adhere to the nomenclature used in the literature, we will keep using s when referring to states, even when the statistics are held in the nodes that (in open loop) do not store the state s.

$$R_{max}(s,a) = max_{a' \in A(s')} R(s', a') \quad (4)$$

The weight w_r has been set to 0.5 by trial and error. This policy gives more weight to those states nearer in the future, and it is inspired by the One-pass Exponential Weighted Average [10].

4.3 Tree Search Approaches

Two different tree search approaches have been used in the experimental study:

- **DBS:** A Directed Breadth First Search (DBS) algorithm has been implemented as our first tree search approach. The objective of this simple algorithm is to serve as a baseline controller for comparison with the other two. Starting from the current state, a child node is created for each one of the available actions. The tree is then traversed depth level after depth level until a terminal game state is reached, or the time budget runs out. The tree search is *directed* because it evaluates the children in order, according to the highest average reward $Q(s,a)$. When this process finishes, the action to take in the game is selected by the recommendation policy described in Section 4.2.
- **UCB1-TS:** The second tree search approach proposed in this paper uses the UCB1 Equation 1 as tree policy, instead of following the child with the highest average reward. The algorithm grows the game tree iteratively up to a determined simulation depth D, adding a new node to the tree at each iteration. All states found during the navigation of the tree are evaluated with the heuristic described in section 4.1. When the time budget is exhausted, the move to make is chosen by the recommendation policy from Section 4.2.

4.4 RHEA

As with the tree search algorithms, it is important that the RHEA is implemented in an *anytime* fashion (i.e. that it can be stopped at any time, yielding a valid solution). In order to avoid having long evaluation times, where a whole population is evaluated at once, the algorithm proposed here evaluates one new individual per generation. If this new individual is found better than the worse one of the population, the latter is replaced by the former. This assures that the best individuals are always kept in the population.

The population has a size T, and its individuals are encoded as strings of actions with a determined length L, which establishes how far into the future the plan is performed. Generation of new individuals is performed in two different ways: by the recombination of two different individuals from the population, or by the mutation of one of them. Recombination of two individuals is performed by applying uniform crossover. Mutation replaces one gene of the individual, chosen uniformly at random, with a new value, also determined randomly. A parameter α defines the probability of choosing one of the two breeding methods at each generation.

In both cases, the selection of individuals is performed by an approach similar to non-linear ranking [18], but using a Gaussian function instead of a Polynomial one. All individuals of the population are ranked according to their fitness value, and selected for reproduction by using Gauss Selection [7]. Two different selection mechanisms were tried in a pre-experimental process:

- Normal Gauss Selection: A random value is drawn from a Gauss distribution and mapped to the ranked population.
- P-Gauss Selection: With probability $P = 0.5$, the best individual of the population is selected. Otherwise, a normal Gauss Selection is employed.

After some initial testing, P-Gauss Selection proved to provide better solutions, and therefore it was chosen for the experiments detailed in Section 5.

Fitness values are calculated by executing the sequence of actions of the individual using the forward model, until all actions are executed or a final game state is reached. While evaluating an individual, a tree is generated with the actions performed. When a new sequence of actions is executed from the current state (i.e. from the root of the tree), new nodes will be added to the tree accordingly. When some of these sequences are repeated in different individuals, the tree nodes will be reused to gather the stored statistics. Therefore, the evaluation of the individuals of the population is used to generate a game tree, similar to the ones employed by the tree search approaches. This provides the algorithms with a way of calculating statistics about the actions taken, as well as reusing information from one game step to the next by keeping the game tree, as described before in Figure 1.

The fitness function proposed in this paper maximizes the average of the $Q(s,a)$ values of all nodes visited during the evaluation of the individual. This allows not only the state of the game reached after the sequence of actions has been executed to be taken into account, but also valuing intermediate states. Alternative options, such as using only the reward of the state reached at the end of the evaluation, would not take advantage of the statistics stored in the game tree: note that the last state is only visited more than once if two or more individuals with exactly the same sequence of actions are evaluated.

5. EXPERIMENTAL STUDY

5.1 Experimental Setup

A thorough experimental study has been performed to analyze the quality of the algorithms presented in this paper. For the RHEA approach, the parameters tested are the breeding parameter ($\alpha = \{0.00, 0.25, 0.50, 0.75, 1.00\}$), the individual length ($L = 2$ to 7) and the population size ($T = \{5, 20, 100\}$).

For the UCB1-TS, the parameters tested are the simulation depth ($D = 1$ to 7) and the value of the constant C for UCB1 (see Equation 1; $C = \{0.00, 0.33, 0.67, 1.00, 1.33, 1.67, 2.00\}$).

All configurations have been tested 10 times on each one of the 5 levels of the 29 games from the training set of the GVG-AI framework (Intel Xeon CPU E3-1245, 3.40GHz and 32GB of memory). Therefore, each algorithm has played 1450 games, with the exception of DBS, which has played each level 100 times (thus playing a total of 14500 games), and has no parameters to tune.

Finally, the DBS controller and the best configurations found for RHEA and UCB1-TS, have been submitted to the GVG-AI competition website for its evaluation in the test set (that contains 10 unknown games) in order to compare these approaches with other entries from the contest. According to the rules of the competition, the server executes each controller 10 times on each one of the 5 levels of the 10 games from this set.

5.2 Results on the Training Set

Table 1 shows the average (with standard error of the averages) of the percentage of games won by UCB1-TS, for the configurations tested for this algorithm. As can be seen, higher values of the UCB1 constant C seem to provide a higher rate of victories. The right-most column shows an increment of this measure as C approaches to 2.00. This implies that a higher exploration in the search space tends to provide better results. Actually, the worst average of results is obtained with $C = 0.00$, which effectively

		Simulation Depth D							
		1	2	3	4	5	6	7	Avg.
UCB1 Constant C	0.00	36.23 (0.54)	38.62 (0.38)	38.97 (0.19)	38.62 (0.53)	38.90 (0.48)	38.90 (0.28)	39.03 (0.33)	38.47 (0.17)
	0.33	37.32 (0.1)	40.75 (0.48)	44.93 (0.38)	44.58 (0.3)	44.24 (0.34)	44.86 (0.26)	44.10 (0.32)	42.97 (0.14)
	0.67	35.68 (0.49)	39.86 (0.27)	44.72 (0.34)	44.72 (0.44)	43.63 (0.33)	44.10 (0.39)	43.49 (0.12)	42.31 (0.17)
	1.00	36.57 (0.31)	40.20 (0.3)	45.48 (0.4)	44.31 (0.31)	42.87 (0.37)	43.76 (0.47)	44.31 (0.44)	42.50 (0.15)
	1.33	36.78 (0.37)	40.68 (0.27)	44.79 (0.14)	44.79 (0.27)	44.17 (0.44)	43.42 (0.34)	43.69 (0.29)	42.62 (0.12)
	1.67	35.68 (0.52)	40.61 (0.33)	44.79 (0.16)	44.86 (0.36)	43.08 (0.45)	43.76 (0.38)	44.04 (0.41)	42.40 (0.15)
	2.00	35.81 (0.48)	40.27 (0.3)	**46.16** **(0.35)**	45.54 (0.28)	43.49 (0.36)	44.04 (0.26)	43.69 (0.26)	42.71 (0.13)
	Avg.	36.30 (0.54)	40.14 (0.38)	44.26 (0.32)	43.92 (0.43)	42.91 (0.46)	43.26 (0.42)	43.19 (0.44)	42.00 (0.35)

Table 1: Average (Standard Error) of the percent of games won by UCB1-TS. Best result in bold font.

		Breeding parameter α					
		0.00	0.25	0.50	0.75	1.00	Avg.
Individual length L	2	24.65 (0.32)	41.36 (0.71)	40.13 (0.49)	38.97 (0.36)	40.20 (0.15)	37.06 (0.25)
	3	26.78 (0.45)	42.39 (0.54)	43.35 (0.76)	42.73 (0.59)	**44.59** **(0.47)**	39.97 (0.35)
	4	29.38 (0.36)	43.35 (0.33)	44.10 (0.32)	44.24 (0.19)	44.04 (0.39)	41.02 (0.2)
	5	30.27 (0.48)	42.05 (0.51)	42.53 (0.34)	42.60 (0.65)	42.39 (0.58)	39.97 (0.32)
	6	28.15 (0.36)	42.94 (0.37)	43.15 (0.53)	43.90 (0.45)	43.42 (0.37)	40.31 (0.26)
	7	29.31 (0.68)	41.50 (0.65)	42.60 (0.57)	43.08 (0.41)	42.60 (0.56)	39.82 (0.36)
	Avg.	28.09 (0.57)	42.27 (0.67)	42.64 (0.65)	42.59 (0.57)	42.87 (0.54)	39.69 (0.26)

Table 2: Average (Standard Error) of the percent of games won by RHEA ($T = 5$). Best result in bold font.

		Breeding parameter α					
		0.00	0.25	0.50	0.75	1.00	Avg.
Individual length L	2	35.34 (0.41)	41.37 (0.29)	39.79 (0.43)	39.86 (0.56)	39.52 (0.43)	39.17 (0.26)
	3	39.31 (0.57)	44.93 (0.45)	44.99 (0.55)	44.86 (0.43)	44.04 (0.35)	43.63 (0.39)
	4	38.21 (0.25)	44.10 (0.55)	45.27 (0.53)	45.41 (0.26)	46.36 (0.34)	43.87 (0.24)
	5	38.49 (0.41)	**47.26** **(0.45)**	46.37 (0.16)	45.68 (0.59)	47.05 (0.3)	44.97 (0.24)
	6	36.50 (0.49)	44.93 (0.58)	45.41 (0.39)	45.96 (0.46)	46.98 (0.44)	43.95 (0.29)
	7	36.30 (0.61)	46.23 (0.21)	44.31 (0.4)	44.59 (0.34)	45.27 (0.38)	43.34 (0.24)
	Avg.	37.36 (0.59)	44.80 (0.54)	44.36 (0.53)	44.39 (0.57)	44.87 (0.48)	43.16 (0.24)

Table 3: Average (Standard Error) of the percent of games won by RHEA ($T = 20$). Best result in bold font.

means that the tree policy follows greedily the best child of a node at each movement within the tree.

Regarding the simulation depth (D), there seems to be an optimal spot (among the values tested for D) at $D = 3$, with a slightly worse rate of victories when this value is increased, and definitely worse results when D is set to 1 or 2. This result suggests that an intermediate number of steps from the current state (not too short, not too long) produces higher rates of victories in the 29 games tested. Note that the longer the simulation depth, the smaller the amount of iterations that can be performed by UCB1-TS in the available time budget. Therefore, an intermediate value of D conveys a good equilibrium between how long the algorithm simulates into future states and the number of iterations, which give a higher confidence in the statistics stored in the nodes of the tree.

The best average of victories obtained by UCB1-TS corresponds to the best values of these two parameters, providing a rate of 46.16% of victories, with a standard error of 0.35. For a comparison with the other algorithms, the percentage of games won by UCB1-TS considering all configurations tested is 42.00% with a standard error of 0.35.

Tables 2, 3 and 4 show the averages (with standard errors) of each configuration tested for the RHEA approach, employing population sizes of 5, 20 and 100, respectively. There are some general conclusions that can be drawn by analyzing the results obtained by this algorithm.

The first aspect worth noticing is that a small population produces lower victory rates, with averages when $T = 5$ around 3 or 4 points lower than population sizes of 20 and 100. The results between $T = 20$ and $T = 100$ are similar, with a slightly higher

	Breeding parameter α					
Individual length L	0.00	0.25	0.50	0.75	1.00	Avg.
2	41.43 (0.33)	40.95 (0.39)	40.00 (0.36)	40.55 (0.60)	39.79 (0.25)	40.54 (0.24)
3	44.99 (0.49)	44.79 (0.45)	45.34 (0.31)	45.82 (0.27)	46.09 (0.33)	45.41 (0.23)
4	43.22 (0.32)	45.13 (0.08)	46.91 (0.51)	46.78 (0.35)	**47.39** **(0.3)**	45.89 (0.19)
5	40.68 (0.55	44.38 (0.37)	46.64 (0.38)	46.36 (0.37)	46.44 (0.41)	44.90 (0.26)
6	39.52 (0.35)	44.11 (0.51)	45.48 (0.52)	45.41 (0.44)	45.54 (0.43)	44.01 (0.28)
7	38.76 (0.34)	43.90 (0.51)	43.15 (0.23)	42.94 (0.33)	43.56 (0.34)	42.46 (0.22)
Avg.	41.43 (0.51)	43.88 (0.49)	44.56 (0.50)	44.64 (0.51)	44.80 (0.44)	43.87 (0.21)

Table 4: Average (Standard Error) of the percent of games won by RHEA ($T = 100$). Best result in bold font.

victory rate for $T = 100$. This again suggests that a higher exploration of the search space is better for the controller.

Limiting ourselves to analyzing the results obtained for $T = 20$ and 100, it can be observed that a good individual length is again (as it happened with UCB1-TS) a good compromise between how far a sequence of actions simulates into the future and the amount of evaluations that can be performed during a game cycle (longer sequences imply that fewer individuals can be evaluated in the time budget). In this case, intermediate values of L such as 5 (for $T =$ 20) and 4 (for $T = 100$) seem to perform best.

Regarding the breeding parameter, the differences on performance are small when $\alpha > 0.00$. A priori, it seemed obvious that a value of $\alpha = 0.00$ would not achieve very good results, as the only way of creating new individuals does not generate any new genetic material by mutation. However, it is still interesting to analyze the results obtained. For $T = 100$, the results are around 10 points better when $\alpha = 0.00$ in $T = 5$, and close to 4 points higher than in the $T = 20$ case. This is of course due to a higher variety of individuals in a population with a larger size.

Nevertheless, it is clear that setting α to 0.00 is not an optimal strategy, because the best results are found when $\alpha = 0.25$ and $L = 5$ for $T = 20$ (47.26 with a standard error of 0.45), and $\alpha = 1.00$ and $L = 4$ for $T = 100$ (47.39 with a standard error of 0.3). The average of victories in all experiments run with the RHEA approach is 42.2 (0.42).

The experiments performed with DBS provide an average of 39.0 (0.38) of games won, significantly lower than the other two approaches. Table 5 compares the results of all algorithms tested in this study, showing the average (with standard error), minimum and maximum rates of victories of all runs of each approach. Both UCB1-TS and RHEA are clearly better than DBS, with the evolutionary approach slightly ahead of the best tree search technique in terms of average of victories.

Finally, in a game per game basis, the results suggest that the games where the proposed techniques fail more often are those where the rewards are delayed far in the future.

5.3 Results on the Test Set

The top configurations of each controller have been submitted for testing to the GVG-AI competition's website server (Intel Core i5, 2.90GHz, and 4GB of memory), so they can be ranked according to the contest rules as described at the end of Section 2. This ranking awards those controllers that play better across the set of 10 unknown games of the competition. The controllers submitted are DBS (its *username* in the server - and in Figure 3 - is *TeamTopbug_HR*), UCB1-TS (*TeamTopbug_RL*) with $D = 3$ and $C = 2.00$, and RHEA with $T = 100$, $L = 4$ and $\alpha = 1.00$ (*TeamTopbug_NI*).

	Min	Average (Std. Err)	Max
DBS	36.3	39.0 (0.38)	41.1
UCB1-TS	35.7	42.0 (0.35)	46.2
RHEA	24.7	42.2 (0.42)	47.4

Table 5: Percent of games won by DBS, UCB1-TS and RHEA.

Figure 3 shows the three algorithms in the rankings with the first entries submitted to the competition (for the sake of space, the bottom 15 entries are not included in this figure). As can be seen, the RHEA (*TeamTopbug_NI*) approach obtains more points than the UCB1-TS algorithm, getting a better result in 9 of the 10 games of the set. The evolutionary approach obtains 44.8% of victories in this game, while the two tree approaches get around 41.0%. The winner of the 2014 competition, still the first entry at the time of writing of this paper, achieves a rate of victories of 51.2%.

Compared to the other entries of the competition, our evolutionary approach ranks 4^{th}, obtaining the best overall result in two games of the set. It is worthwhile highlighting that our RHEA controller is the best evolutionary approach found in the rankings; the other entries in Figure 3 are variants of MCTS, the second best EA controller is ranked 14^{th}, and the sample Genetic Algorithm (GA) controller supplied with the framework is in 18^{th} position.

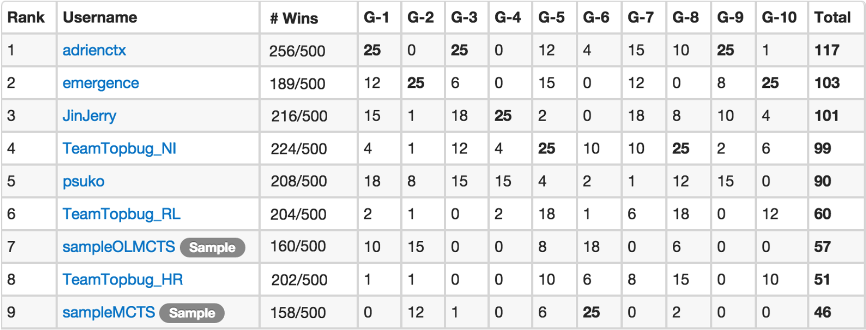
6. CONCLUSIONS AND FUTURE WORK

This paper presents several approaches for general video game playing in real-time games. Concretely, it proposes two tree search algorithms and a RHEA combined with the generation of a game tree search. All these approaches are open loop planning algorithms, as they do not rely on specific states found by the forward model, but on the statistics gathered after executing sequences of actions. This paper also proposes a general purpose heuristic, with no game-dependent knowledge, employed to play up to 39 different real-time games.

The results obtained in this paper allow several conclusions to be drawn about the algorithms tested. First, regarding the tree search techniques, the experimental study shows the advantage of using a tree policy that finds a balance between exploration of the search space and exploitation of the best action found so far, tested across a large set of games. The proposed RHEA algorithm obtains very good results in the set of unknown 10 games of the GVG-AI competition, becoming the highest ranked evolutionary approach. All algorithms tested rank in the top third of the rankings, which suggests that the common heuristic employed (that maximizes level exploration) works well across algorithms and games.

Potential future work of this research may include exploration of other values for the parameters tested. For instance, initial tests show that larger values of simulation depth ($\sim$ 20) behave extremely well in some of the games tested, although more thorough experimental work is needed to analyze this.

Additionally, other techniques found in the literature may help to further improve the results obtained in this research, such as the



Rank	Username	# Wins	G-1	G-2	G-3	G-4	G-5	G-6	G-7	G-8	G-9	G-10	Total
1	adrienctx	256/500	**25**	0	**25**	0	12	4	15	10	**25**	1	**117**
2	emergence	189/500	12	**25**	6	0	15	0	12	0	8	**25**	**103**
3	JinJerry	216/500	15	1	18	**25**	2	0	18	8	10	4	**101**
4	TeamTopbug_NI	224/500	4	1	12	4	**25**	10	10	**25**	2	6	**99**
5	psuko	208/500	18	8	15	15	4	2	1	12	15	0	**90**
6	TeamTopbug_RL	204/500	2	1	0	2	18	1	6	18	0	12	**60**
7	sampleOLMCTS Sample	160/500	10	15	0	0	8	18	0	6	0	0	**57**
8	TeamTopbug_HR	202/500	1	1	0	0	10	6	8	15	0	10	**51**
9	sampleMCTS Sample	158/500	0	12	1	0	6	**25**	0	2	0	0	**46**

Figure 3: Results of the three algorithms in the test set of the GVG-AI 2014 Competition.

use of macro-actions [15] to reduce the search space and effectively enjoy a longer time budget for action decision. Another possibility is to address the problem with Multi-Objective Optimization techniques (both evolutionary and tree-search based) [14]. An approach that takes the victory condition, the maximization of the game score and the exploration of the level as separate but complimentary objectives, seems to be a viable alternative.

7. REFERENCES

[1] R. Beer and J. Gallagher. Evolving Dynamical Neural Networks for Adaptive Behavior. *Adaptive behavior*, 1(1):91–122, 1992.

[2] C. Browne, E. J. Powley, D. Whitehouse, S. Lucas, P. I. 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.

[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*, pages 216–217, 2006.

[4] M. Ebner, J. Levine, S. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a Video Game Description Language. *Dagstuhl Follow-up*, 6:85–100, 2013.

[5] H. Finnsson and Y. Björnsson. CADIA-Player: A Simulation Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1:1–12, 2009.

[6] M. Genesereth, N. Love, and B. Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26:62–72, 2005.

[7] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. M. Kaufmann, 1991.

[8] F. Gomez and R. Miikkulainen. Solving Non-Markovian Control Tasks with Neuroevolution. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1356–1361. Lawrence Erlbaum Associates LTD, 1999.

[9] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. *Machine Learning: ECML*, 4212:282–293, 2006.

[10] J. Loveless, S. Stoikov, and R. Waeber. Online Algorithms in High-Frequency Trading. *Comm. ACM*, 56(10):50–56, 2013.

[11] S. Lucas and G. Kendall. Evolutionary Computation and Games (Invited Review). *IEEE Computational Intelligence Magazine*, 1(1):10–18, February 2006.

[12] J. Méhat and T. Cazenave. A Parallel General Game Player. *KI - Künstliche Intelligenz*, 25:43–47, 2011.

[13] T. Nielsen, G. Barros, J. Togelius, and M. Nelson. General Video Game Evaluation using Relative Algorithm Performance Profiles. In *EvoApplications*, 2015.

[14] D. Perez, S. Mostaghim, S. Samothrakis, and S. Lucas. Multi-Objective Monte Carlo Tree Search for Real-Time Games. *IEEE Transactions on Computational Intelligence and AI in Games*, pp:1–13, DOI: 10.1109/TCIAIG.2014.2345842, 2014.

[15] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, and S. Lucas. Solving the Physical Travelling Salesman Problem: Tree Search and Macro-Actions. *IEEE Transactions on Computational Intelligence and AI in Games*, 6:1:31–45, 2013.

[16] 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 Conference on Genetic and Evolutionary Computation (GECCO)*, pages 351–358, 2013.

[17] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, page (to appear) DOI: 10.1109/TCIAIG.2015.2402393, 2015.

[18] H. Pohlheim. Ein genetischer Algorithmus mit Mehrfachpopulationen zur numerischen Optimierung. *at-Automatisierungstechnik*, 43(3):127–135, 1995.

[19] S. J. Russell. Rationality and Intelligence. *Artificial Intelligence*, 1:57–77, 1997.

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

[21] S. Samothrakis and S. Lucas. Planning Using Online Evolutionary Overfitting. In *UK Workshop on Computational Intelligence*, pages 1–6. IEEE, 2010.

[22] T. Schaul. A Video Game Description Language for Model-based or Interactive Learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 193–200, Niagara Falls, 2013. IEEE Press.

[23] R. Weber. *Optimization and Control*. University of Cambridge, 2010.

[24] W. Zhe, K. Q. Nguyen, R. Thawonmas, and F. Rinaldo. Adopting Scouting and Heuristics to Improve the AI Performance in Starcraft. *Proceedings of the Innovations in Information and Communication Science and Technology IICST 2013*, pages 155–164, Sept. 2013.