View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Essex Research Repository

Knowledge-based Fast Evolutionary MCTS for General Video Game Playing

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

***Abstract*—General Video Game Playing is a game AI domain in which the usage of game-dependent domain knowledge is very limited or even non existent. This imposes obvious difficulties when seeking to create agents able to play sets of different games. Taken more broadly, this issue can be used as an introduction to the field of General Artificial Intelligence. This paper explores the performance of a vanilla Monte Carlo Tree Search algorithm, and analyzes the main difficulties encountered when tackling this kind of scenarios. Modifications are proposed to overcome these issues, strengthening the algorithm's ability to gather and discover knowledge, and taking advantage of past experiences. Results show that the performance of the algorithm is significantly improved, although there remain unresolved problems that require further research. The framework employed in this research is publicly available and will be used in the General Video Game Playing competition at the IEEE Conference on Computational Intelligence and Games in 2014.**

I. Introduction

Games are splendid benchmarks to test different algorithms, understand how new add-ons or modifications affect their performance, and to compare different approaches. Quite often, the research exercise is focused on one or several algorithms, measuring their performance in one particular game.

In most games used by researchers, there exists the possibility of adding a significant amount of domain knowledge that helps the algorithms tested to achieve the game goals. Whilst not considered 'incorrect', this addition can raise doubts as to whether the type of knowledge introduced is better suited to only certain of the algorithms under comparison, or even cast doubt on how much of the success of an algorithm can be attributed to the algorithm itself or to the heuristics employed.

It could even be possible to argue that, in spite of applying the same heuristics in all algorithms, the comparison could still be unbalanced in certain cases, as some heuristics could benefit certain algorithms more than others. When comparing algorithms, it may possibly be fairer to allow each algorithm to use its best possible heuristic. But should we not then compare the quality of each heuristic separately as well?

The objective of General Game Planning (GGP) and General Video Game Playing (GVGP) is to by-pass the addition of game specific knowledge, especially if the algorithm is tested in games that have not been played before. This is the aim of the General Video Game Playing Competition [15], a computer game playing contest that will be held for the first time at the IEEE Conference on Computational Intelligence and Games (CIG) 2014.

Obviously, algorithms that approach GVGP problems may still count on some kind of domain knowledge, and the questions raised above could still be asked. Indeed, many different algorithms can be employed for GVGP, and chances are that heuristics will still make a big difference. However, by reducing the game-dependent knowledge, approaches are forced to be more general, and research conducted in this field is closer to the open domain of General Artificial Intelligence.

The goal of this paper is to show how and why a well known algorithm, Monte Carlo Tree Search (MCTS), struggles in GVGP when there is no game specific information, and to offer some initial ideas on how to overcome this issue.

The paper is structured as follows: Section II reviews the related literature in GGP and GVGP, as well as past uses of MCTS in this field. Then, Section III describes the framework used as a benchmark for this research. Section IV explains the MCTS algorithm, a default controller and its limitations. Section V proposes a solution to the problems found in the previous section. Section VI details the experimental work and, finally, Section VII draws some conclusions and proposes future work.

II. Related Research

One of the first attempts to develop and establish a general game playing framework was carried out by the Stanford Logic Group of Stanford University, when they organized the first AAAI GGP competition in 2005 [8]. In this competition, players (also referred to in this paper as *agents*, or *controllers*) would receive declarative descriptions of games at runtime (hence, the game rules were unknown beforehand), and would use this information to play the game effectively. These competitions feature finite and synchronous games, described in a Game Definition Language (GDL), and include games such as Chess, Tic-tac-toe, Othello, Connect-4 or Breakthrough.

Among the winners of the different editions, Upper Confidence bounds for Trees (UCT) and Monte Carlo Tree Search (MCTS) approaches deserve a special mention. CADIA-Player, developed by Hilmar Finnsson in his Master's thesis [5], [6], was the first MCTS based approach to be declared winner of the competition, in 2007. The agent used a form of historic heuristic and parallelization, capturing game properties during the algorithm roll-outs.

The winner of two later editions, 2009 and 2010, was another MCTS based approach, developed by J. Méhat and

T. Cazenave [12]. This agent, named Ary, studied the concept of parallelizing MCTS in more depth, implementing the *root parallel algorithm*. The idea behind this technique is to perform independent Monte-Carlo tree searches in parallel in different CPU. When the decision time allowed to make a move is over, a master component decides which action to take among the ones suggested by the different trees.

Other interesting MCTS-based players for GGP are *Centurio*, from Möller et al. [13], that combines parallelized MCTS with Answer Set Programming (ASP), and the agent by Sharma et al. [17], that generates domain-independent knowledge and uses it to guide the simulations in UCT.

In GGP, the time allowed for each player to pick a move can vary from game to game. The time allowed is usually indicated in seconds. Therefore, any player will be able to spend at least 1 second of decision time in choosing the next action to make. Whilst this is an appropriate amount of time for the types of games that feature in GGP competitions, a similar decision time cannot be afforded in video (real-time) games. Here the agents perform actions at a much higher rate, making them appear almost continuous to a human player, in contrast to turn-based games. Additionally, the real-time component allows for an asynchronous interaction between the player and the other entities: the game progresses even if the player does not take any action.

During the last few years, important research has been carried out in the field of general video-game playing, specifically in arcade games where the games are clearly not turn-based and where the time allowed for the agents to pick an action is measured in milliseconds. Most of this research has been performed in games from the Atari 2600 collection. Bellemare et al. [2] introduced the Arcade Learning Environment (ALE), a platform and a methodology to evaluate AI agents in domain-independent environments, employing 55 games from Atari 2600. In this framework, each observation consists of a single frame: a two dimensional (160×210) array of 7-bit pixels. The agent acts every 5 frames (12 times per second), and the action space contains the 18 discrete actions allowed by the joystick. The agent, therefore, must analyze the screen to identify game objects and perform a move accordingly, in a decision time of close to 80ms.

Several approaches have been proposed to deal with this kind of environment, such as Reinforcement Learning and MCTS [2]. M. Hausknecht et al. [9] employed evolutionary neural networks to extract higher-dimensional representation forms from the raw game screen. Yavar Naddaf, in his Master's thesis [14], extracted feature vectors from the game screen, that were used in conjunction with Gradient-descent Sarsa(λ) and UCT. Also in this domain, Bellemare et al. [7] explored the concept of contingency awareness using Atari 2600 games. Contingency awareness is "the recognition that a future observation is under an agent's control and not solely determined by the environment" [7, p. 2]. In this research, the authors show that contingency awareness helps the agent to track objects on the screen and improve on existing methods for feature construction.

Similar to the Atari 2600 domain, J. Levine et al. [10] recently proposed the creation of a benchmark for General Video Game playing that complements Atari 2600 in two ways: the creation of games in a more general framework, and the organization of a competition to test the different approaches to this problem. Additionally, in this framework, the agent does not need to analyze a screen capture, as all information is accessible via encapsulated objects. It is also worthwhile highlighting that the game rules are never given to the agent, something that is usually done in GGP.

This new framework is based on the work of Tom Schaul [16], who created a Video Game Description Language (VGDL) to serve as a benchmark for learning and planning problems. The first edition of the General Video Game AI (GVGAI) Competition, organized by the authors of the present paper, and Julian Togelius and Tom Schaul, will be held at the IEEE Conference on Computational Intelligence and Games (CIG) in 2014 [15]. This paper uses the framework of this competition as a benchmark, including some of the games that feature in the contest.

III. The GVGAI Framework

This section describes the framework and games employed in this research.

A. Games

The GVGAI competition presents several games divided into three sets: training, validation, and test. The first set (the only one ready at the time this research was conducted) is public to all competitors, in order for them to train and prepare their controllers, and is the one used in the experiments conducted for this paper. The objective of the validation set is to serve as a hidden set of games where participants can execute their controllers in the server. Finally, the test set is another set of secret games for the final evaluation of the competition.

Each set of games is composed of 10 games, and there are 5 different levels available for each one of them. Most games have a non-deterministic element in the behaviour of their entities, producing slightly different playouts every time the same game and level is played. Also, all games have a maximum number of game steps to be played (2000), in order to avoid degenerate players never finishing the game. If this limit is violated, the result of the game will count as a loss.

The 10 games from the training set are described in Table I. As can be seen, the games differ significantly in winning conditions, different number of non-player characters (NPCs), scoring mechanics and even in the available actions for the agent. For instance, some games have a timer that finishes the game with a victory (as in *Survive Zombies*) or a defeat (as in *Sokoban*). In some cases, it is desirable to collide with certain moving entities (as in *Butterflies*, or in *Chase*) but, in other games, those events are what actually kill the player (as in *Portals*, or also in *Chase*). In other games, the agent (or *avatar*) is killed if it collides with a given sprite, that may only be killed if the avatar picks the action `USE` appropriately

Game	Description	Score	Actions
Aliens	Similar to traditional Space Invaders, Aliens features the player (avatar) in the bottom of the screen, shooting upwards at aliens that approach Earth, who also shoot back at the avatar. The player loses if any alien touches it, and wins if all aliens are eliminated.	• 1 point is awarded for each alien or protective structure destroyed by the avatar. • -1 point is given if the player is hit.	LEFT, RIGHT, USE.
Boulderdash	The avatar must dig in a cave to find at least 10 diamonds, with the aid of a shovel, before exiting through a door. Some heavy rocks may fall while digging, killing the player if it is hit from above. There are enemies in the cave that might kill the player, but if two different enemies collide, a new diamond is spawned.	• 2 points are awarded for each diamond collected, and 1 point every time a new diamond is spawned. • -1 point is given if the avatar is killed by a rock or an enemy.	LEFT, RIGHT, UP, DOWN, USE.
Butterflies	The avatar must capture butterflies that move randomly around the level. If a butterfly touches a cocoon, more butterflies are spawned. The player wins if it collects all butterflies, but loses if all cocoons are opened.	• 2 points are awarded for each butterfly captured.	LEFT, RIGHT, UP, DOWN.
Chase	The avatar must chase and kill scared goats that flee from the player. If a goat finds another goat's corpse, it becomes angry and chases the player. The player wins if all scared goats are dead, but it loses if is hit by an angry goat.	• 1 point for killing a goat. • -1 point for being hit by an angry goat.	LEFT, RIGHT, UP, DOWN.
Frogs	The avatar is a frog that must cross a road, full of tracks, and a river, only traversable by logs, to reach a goal. The player wins if the goal is reached, but loses if it is hit by a truck or falls into the water.	• 1 point for reaching the goal. • -2 points for being hit by a truck.	LEFT, RIGHT, UP, DOWN.
Missile Command	The avatar must shoot at several missiles that fall from the sky, before they reach the cities they are directed towards. The player wins if it is able to save at least one city, and loses if all cities are hit.	• 2 points are given for destroying a missile. • -1 point for each city hit.	LEFT, RIGHT, UP, DOWN, USE.
Portals	The avatar must find the goal while avoiding lasers that kill him. There are many portals that teleport the player from one location to another. The player wins if the goal is reached, and loses if killed by a laser.	• 1 point is given for reaching the goal.	LEFT, RIGHT, UP, DOWN.
Sokoban	The avatar must push boxes so they fall into holes. The player wins if all boxes are made to disappear, and loses when the timer runs out.	• 1 point is given for each box pushed into a hole.	LEFT, RIGHT, UP, DOWN.
Survive Zombies	The avatar must stay alive while being attacked by spawned zombies. It may collect honey, dropped by bees, in order to avoid being killed by zombies. The player wins if the timer runs out, and loses if hit by a zombie while having no honey (otherwise, the zombie dies).	• 1 point is given for collecting one piece of honey, and also for killing a zombie. • -1 point if the avatar is killed, or it falls into the zombie spawn point.	LEFT, RIGHT, UP, DOWN.
Zelda	The avatar must find a key in a maze to open a door and exit. The player is also equipped with a sword to kill enemies existing in the maze. The player wins if it exits the maze, and loses if it is hit by an enemy.	• 2 points for killing an enemy, 1 for collecting the key, and another point for reaching the door with it. • -1 point if the avatar is killed.	LEFT, RIGHT, UP, DOWN, USE.

TABLE I: Games in the training set of the GVGAI Competition, employed in the experiments of this paper.

(in close proximity, as when using the sword in *Zelda*, or at a greater distance, as when shooting in *Aliens*). Figure 1 shows some games from the training set.

These differences in game play make the creation of a simple game-dependent heuristic a relatively complex task, as the different mechanisms must be handled on a game per game basis. Furthermore, a controller created following these ideas, would probably fail to behave correctly in other unseen games in the competition.

B. The framework

All games are written in VGDL, a video-game description language that is able to fully define a game, typically in less than 50 lines. For a full description of VGDL, the reader should consult [16]. The framework used in the competition, and in this paper, is a Java port of the original Python version of VGDL, originally developed by Tom Schaul.

Each controller in this framework must implement two methods, a constructor for initializing the agent (only called once), and an `act` function to determine the action to take at every game step. The real-time constraints of the framework determine that the first call must be completed in 1 second, while every `act` call must return a move to make within a time budget of 40 milliseconds (or the agent will be disqualified). Both methods receive a timer, as a reference, to know when the call is due to end, and a `StateObservation` object representing the current state of the game.

The state observation object is the window the agent has to the environment. This state can be *advanced* with a given action, allowing simulated moves by the agent. As the games are generally stochastic, it is the responsibility of the agent to determine how to trust the results of the simulated moves.

The `StateObservation` object also provides information about the state of the game, such as the current time step,

Fig. 1: Four of the ten training set games: from top to bottom, left to right, *Boulderdash*, *Survive Zombies*, *Aliens* and *Frogs*.

score, or whether the player won or lost the game (or is still ongoing). Additionally, it provides the list of actions available in the game that is being played. Finally, two more involved pieces of information are available to the agent:

- A history of avatar events, sorted by time step, that have happened in the game so far. An avatar event is defined as a collision between the avatar, or any sprite produced by the avatar (such as bullets, shovel or sword), and any other sprite in the game.
- Lists of observations and distances to the sprites in the game. One list is given for each sprite type, and they are grouped by the sprite's *category*. This category is determined by the *apparent* behaviour of the sprite: static, non-static, NPCs, collectables and portals (doors).

Although it might seem that the latter gives too much information to the agent, the controller still needs to figure out how these different sprites affect the game. For instance, no information is given as to whether the NPCs are friendly or dangerous. The agent does not know if reaching the portals would make him win the game, would kill the avatar, or simply teleport it to a different location within the same level.

IV. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a tree search algorithm that has had an important impact in Game AI since it was introduced in 2006 by several researchers. An extensive survey of MCTS methods is covered by Browne et al. in [4].

MCTS estimates the average value of rewards by iteratively sampling actions in the environment, building an asymmetric tree that leans towards the most promising portions of the search space. Each node in the tree holds certain statistics about how often a move is played from that state ($N(s,a)$), how many times that node is reached ($N(s)$) and the average reward ($Q(s,a)$) obtained after applying a move a in the state s. On each iteration, or play-out, actions are simulated from the root until either the end of the game or a maximum simulation depth is reached. Figure 2 shows the four steps of each iteration.

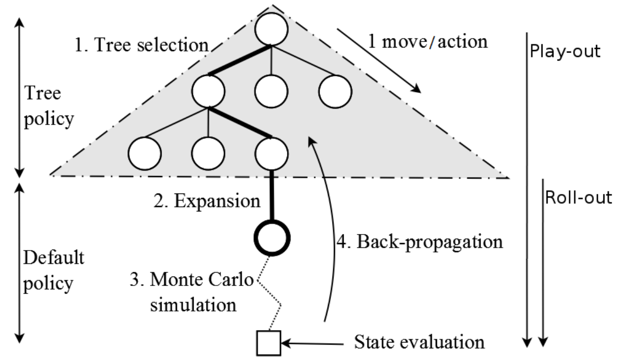

Fig. 2: MCTS algorithm steps.

When the algorithm starts, the tree is formed only by the root node, which represents the current state of the game. In the first stage, *Tree selection*, the algorithm navigates through the tree until it reaches a node that is not fully expanded (this represents one of the tree's children that has never been explored). On each one of these selections, MCTS balances between exploration (actions that lead to less explored states) and exploitation (choosing the action with the best estimated reward). This is known as the MCTS *tree policy*, and one of the typical ways this is performed in MCTS is by using Upper Confidence Bounds (UCB1):

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

The balance between exploration and exploitation can be tempered by modifying C. Higher values of C give added weight to the second term of the UCB1 Equation 1, giving preference to those actions that have been explored less, at the expense of taking actions with the highest average reward $Q(s,a)$. A commonly used value is $\sqrt{2}$, as it balances both facets of the search when the rewards are normalized between 0 and 1.

In the second phase, *Expansion*, a new node is added to the tree and the third stage, *Monte Carlo simulation*, is started. Random actions, either uniformly or biased, are taken up to the end of the play-out, where the state is analyzed and given a score or reward. This is known as the MCTS *default policy* (and a *roll-out* is defined as the sequence of actions taken in the *Monte Carlo simulation* step). In the final phase, *Back propagation*, the reward is back propagated through all visited nodes up to the root, updating the stored statistics $N(s,a)$, $N(s)$ and $Q(s,a)$.

One of the main advantages of MCTS is that it is considered to be an *anytime* algorithm. This means that the algorithm may stop at any number of iterations and provide a reasonable, valid, next action to take. This makes MCTS a particularly good choice for real-time games, where the time budget to decide the next move is severely limited.

Once all iterations have been performed, MCTS returns the next action the agent must take, usually according to the statistics stored in the root node. Examples of these policies include taking the action chosen more often (a for the highest $N(s,a)$), the one that provides a highest average reward ($Q(s,a)$), or simply to apply Equation 1 at the root node.

A. SampleMCTS controller

The vanilla MCTS algorithm features in the GVGAI framework as a sample controller. For this controller, the maximum depth of the tree is 10 actions, $C = \sqrt{2}$ and the score of each state is calculated as a function of the game score, normalized between the minimum and maximum scores ever seen during the play-outs. In case the game is won or lost in a given state, the reward is a large positive or negative number, respectively.

B. Analysis

Although detailed results of the sample controller are reported later in Section VI, it is important to analyze first why this controller only achieves a 21.6% victory rate.

Initially, the obvious reason for this low rate of success is that the algorithm has no game specific information to bias the roll-outs, and the rewards depend only on the score and outcome of the game. Additionally, due to the real-time nature of the framework, roll-outs are limited in depth and therefore the vast majority of the play-outs do not reach an end game state, preventing the algorithm from finding winning states. This is, however, a problem that is not possible to avoid: play-out depth is always going to be limited, and the Monte Carlo simulations cannot use any game specific information to bias the actions taken in this domain.

There is another important problem that MCTS faces continuously when playing general video-games: in its vanilla form, MCTS has no way of reusing past information from events that provided some score gain in the past, even during the same game. Imagine that in the game *Butterflies* (see Table I), the avatar has successfully captured some butterflies in any area of the level. Here, MCTS learned that certain actions (that caused the avatar to collide with a butterfly) provided a boost in the score, indirectly learning that colliding with butterflies was a good thing to do. However, these learnt facts are not used again to drive the agent to capture other butterflies that are beyond the horizon reachable by MCTS during the roll-outs.

Furthermore, there is a second important problem: some sprites in the game are never reached by the algorithm's simulations. Imagine the game of *Zelda*, where the avatar must collect a key before exiting the level to win the game. If the key is beyond the simulation horizon (in this case, if it is more than 10 actions away), there is no incentive for the agent to collect it. Note that, if we were programming an agent that would only play this game, an obvious heuristic would be to reduce the distance to the key. But in general video-game playing there is no way to know (*a priori*) what sprites should be targeted by the avatar in the first place.

The next section focuses on a proposed solution to these problems, by providing the algorithm with a knowledge base, and biasing the Monte Carlo simulations to maximize the knowledge gain obtained during the play-outs.

V. Knowledge-based Fast Evolutionary MCTS

Several authors [1], [3] have evolved heuristics, in an offline manner, to bias roll-outs in MCTS. More recently, Lucas et al. [11] proposed an MCTS approach that uses evolution to adapt to the environment and increase performance. In this approach, a vector of weights w is evolved online to bias the Monte Carlo simulations, using a fixed set of features extracted for the current game state. This section proposes an extension of this work, by first employing any number of features, and then dynamically creating a knowledge base that is used to better calculate the reward of a given state.

A. Fast Evolutionary MCTS

The idea behind Fast Evolutionary MCTS is to embed the algorithm roll-outs within evolution. Every roll-out evaluates a single individual of the evolutionary algorithm, providing as fitness the reward calculated at the end of the roll-out. Its pseudocode can be seen in Algorithm 1.

The call in line 4 retrieves the next individual, or weight vector w, to evaluate, while its fitness is set in line 9. The vector w is used to bias the roll-out (line 7), following the next process: mapping from state space S to feature space F. A number of N features are extracted on each state found during the roll-out. Given a set of A available actions, the relative strength of each action (a_i) is calculated as a weighted sum of feature values, as shown in Equation 2.

Algorithm 1 Fast Evolutionary MCTS Algorithm, from [11], assuming one roll-out per fitness evaluation.

```
1: Input: v_0 root state.
2: Output: weight vector w, action a
3: while within computational budget do
4:     w = EVO.GETNEXT()
5:     Initialize Statistics Object S
6:     v_l = TREEPOLICY(v_0)
7:     δ = DEFAULTPOLICY(s(v_l), D(w))
8:     UPDATESTATS(S, δ)
9:     EVO.SETFITNESS(w, S)
10: return w = EVO.GETBEST()
11: return a = RECOMMEND(v_0)
```

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

The weights, initialized at every game step, are stored in a matrix W, where each entry w_{ij} is the weighting of feature j for action i. These relative action strengths are introduced into a softmax function in order to calculate the probability of selecting each action (see Equation 3). For more details about this algorithm, the reader is referred to [11].

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

In this research, the features extracted from each game state are the euclidean distances to the closest NPC, resource, non-static object and portal (as described in Section III-B).

Note that each one of these features may be composed of more than one distance, as there might be more than one type of NPC, resource, portal, etc. For instance, in the game *Chase*, the first feature would return two distances: to the closest scared and to the closest angry goat. The same amount of features will not be available in every game step of the same game, as there could be sprites that do not exist at some point of the game: such as enemy NPCs that have not been spawned yet, or depleted resources.

Therefore, the number N of features not only varies from game to game, but also varies from game step to game step. Fast Evolutionary MCTS must therefore be able to adapt the weight vector according to the present number of features at each step. While in the original Fast Evolutionary MCTS algorithm the number of features was fixed, here the evolutionary algorithm maps each feature to a particular position in the genome, increasing the length of the individual every time a new feature is discovered.

In this research, as in the original Fast Evolutionary MCTS paper, the evolutionary algorithm used is a $(1+1)$ Evolution Strategy (ES). Albeit a simple choice, it produces good results.

B. Knowledge-based Fast Evolutionary MCTS

Now that there is a procedure in place to guide the roll-outs, the next step is to define a score function that provides solutions to the problems analyzed at the end of Section IV-B. Note that this score function is also the one that defines the fitness for the evolved weight vector w, and ultimately will affect how the roll-outs are to be biased.

In this domain, we define the concept *knowledge base* as the combination of two factors: *curiosity* plus *experience*. The former refers to discovering the effects of colliding with other sprites, while the latter allows the agent to reward those events that provided a score gain. Each piece of knowledge corresponds to one event that, as defined in Section IV-B, represents the collision of the avatar - or a sprite produced by the avatar - with another sprite. Specifically, the knowledge base contemplates only the types of sprites used to extract the features (NPC, resource, non-static object and portal). Each one of these *knowledge items* maintains the following statistics:

- Z_i: number of occurrences of the event i.
- $\overline{x_i}$: average of the score change, calculated as the difference between the game score before and after the event took place. It is important to note that an event does not contain information about the proper score change, this needs to be inferred by the controller. As multiple simultaneous events can trigger a score change, the larger the value of Z_i, the more reliable $\overline{x_i}$ will be.

These statistics are updated every time MCTS makes a move in a roll-out. When each roll-out finishes, the following three values are calculated in the final state:

- Score change ΔR: the difference of the game score between the score value at the beginning and at the end of the play-out.
- Knowledge change $\Delta Z = \sum_{i=1}^{N} \Delta(K_i)$: a measure of *curiosity* that values 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 play-out and Z_{iF} is the value of Z_i at the end of the roll-out.

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

  Essentially, ΔZ will be higher when the roll-outs produce more events. Events that have been rarely seen before will provide higher values, rewarding knowledge gathering from events less triggered in the past.
- Distance change $\Delta D = \sum_{i=1}^{N} \Delta(D_i)$: a measure of change in distance to each sprite of type i. Equation 5 defines the value of $\Delta(D_i)$, where D_{i0} is the distance to the closest sprite of type i at the beginning of the play-out, and D_{iF} is the same distance at the end of the roll-out.

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

  Here, ΔD will be higher if the avatar, in the course of the roll-out, has reduced the distance from unknown sprites

(again, measuring *curiosity*), or from those that provided a score boost in the past (*experience*).

Once these three values have been calculated, the final score for the game state reached at the end of the roll-out is obtained as in Equation 6. Essentially, the reward will be the score difference ΔR, unless $\Delta R = 0$. If this happens, none of the actions during the roll-out were able to change the score of the game, and the reward refers to the other two components. The values of $\alpha = 0.66$ and $\beta = 0.33$ have been determined empirically for this research.

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

To summarize, the new score function prioritizes the actions that lead to a score gain in the MCTS iterations. However, if there is no score gain, more reward will be given to the actions that provide more information to the knowledge base, or that will get the avatar closer to sprites that, by colliding with them in the past, seemed to produce a positive score change.

VI. EXPERIMENTS

The experimental work of this paper has been performed on the 10 games explained in Table I. There are five different levels for each one of the games, with variations on the location of the sprites and, sometimes, with slightly different behaviours of the NPC sprites. The complete set of games and levels can be downloaded from the competition website [15]. Each one of the levels is played 5 times, giving a total of 250 games played for each configuration tested. Four different algorithms have been explored in the experiments:

- **Vanilla MCTS**: the sample MCTS controller from the competition, as explained in Section IV-A.
- **Fast-Evo MCTS**: Fast Evolutionary MCTS, as per Lucas et al. [11], using dynamic adaptation of the number of features, as explained in Section V-A.
- **KB MCTS**: Knowledge-based (KB) MCTS as explained in Section V-B, but employing uniformly random roll-outs in the Simulation phase of MCTS (i.e., no Fast Evolution is used to guide the Monte Carlo simulations).
- **KB Fast-Evo MCTS**: Knowledge-based Fast Evolutionary MCTS, as explained in Section V-B, using both knowledge base and evolution to bias the roll-outs.

The experiments can be analyzed by considering two measures: the percentage of victories achieved and the score earned. As in the competition, it is considered that the former value takes precedence over the rankings (it is more relevant to win the game than to lose it with a higher score). Also, comparing the percentage of victories across all games is more representative than comparing scores, as each game has a completely different score system. However, it is interesting to compare scores on a game by game basis.

According to the total average of victories, *KB Fast-Evo MCTS* leads the comparison with $49.2\% \pm 3.2$ of games won. The other MCTS versions all obtained similar victory rates in the range 20% to 25%. This difference shows that adding both the knowledge base and evolution to bias the roll-outs provides a strong advantage to MCTS, but adding each one of these features separately does not impact the vanilla MCTS algorithm significantly. Regarding the scores, *KB Fast-Evo MCTS* also leads on the average points achieved, with 13.5 ± 1.2 points, against the other algorithms (with results all ranging between 9 and 11 points). Nevertheless, as mentioned before, this particular result must be treated with care as different games vary in their score system. It is therefore more relevant to compare scores on a game by game basis.

Table II shows the average victories and scores obtained in every game. In most games, *KB Fast-Evo MCTS* shows a better performance than *Vanilla MCTS* in both percentage of victories and scores achieved. In some cases, like in *Boulderdash*, the increase in victory percentage is obtained when adding the knowledge base system, while in others, as in *Zelda*, it is the evolution feature that gives this boost. On average in most cases, and also in some specific games such *Missile Commmand* and *Chase*, it is both evolution and the knowledge base that cause the improvement. Special mention must be made of *Aliens*, *Butterflies* and *Chase*, in which the *KB Fast-Evo MCTS* algorithm achieved a very high rate of victories. Similarly, *Aliens*, *Chase* and *Missile Command* show a relevant improvement in average score.

KB Fast-Evo MCTS fails, however, to provide good results in certain games, where little (as in *Sokoban* and *Boulderdash*) or no improvement at all (like in *Survive Zombies* and *Frogs*) is observed compared with *Vanilla MCTS*. The reasons are varied as to why this algorithm does not achieve the results it did in other games. Clearly, one is the fact that the distances between sprites are euclidean, not considering obstacles. Shortest distances (i.e. using A*) would positively affect the performance of the algorithm. This is, however, not trivial: path-finding requires the definition of a navigable space. In the GVGAI framework this can be inferred as empty spaces in most games, but in others (particularly in *Boulderdash*), the avatar moves by digging through *dirt*, creating new paths with each movement (but *dirt* itself is not an obstacle).

Nevertheless, not all problems can be attributed to how the distances are calculated. For instance, in *Sokoban*, where boxes are to be pushed, it is extremely important to consider where the box is pushed from. However, the algorithm considers collisions as non-directional events. It could be possible to include this information (which collisions happened from what direction) in the model, but this would be an unnecessary division for other games, where it is not relevant. Actually, it would increase the number of features considered, causing a higher computational cost for their calculation and a larger search space for the evolutionary algorithm.

Another interesting case to analyze is *Frogs*. In this game, the avatar usually struggles with crossing the road. This road is typically composed of three lanes with many trucks that kill the agent when contacting with it. Therefore, the road needs to be crossed quickly, and most of the roll-outs are unable to achieve this without colliding with a truck. The consequence of this is that most of the feedback retrieved suggests that

	Percentage Victories				Average Score			
Game	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**	**Vanila MCTS**	**Fast-Evo MCTS**	**KB MCTS**	**KB Fast-Evo MCTS**
Aliens	8.0 ± 5.4	4.0 ± 3.9	4.0 ± 3.9	**100.0 ± 0.0**	36.72 ± 0.9	38.4 ± 0.8	37.56 ± 1.0	**54.92 ± 1.6**
Boulderdash	0.0 ± 0.0	4.0 ± 3.9	**28.0 ± 9.0**	**16.0 ± 7.3**	9.96 ± 1.0	12.16 ± 1.2	**17.28 ± 1.7**	**16.44 ± 1.8**
Butterflies	88.0 ± 6.5	**96.0 ± 3.9**	80.0 ± 8.0	**100.0 ± 0.0**	27.84 ± 2.8	31.36 ± 3.4	31.04 ± 3.4	28.96 ± 2.8
Chase	12.0 ± 6.5	12.0 ± 6.5	0.0 ± 0.0	**92.0 ± 5.4**	4.04 ± 0.6	4.8 ± 0.6	3.56 ± 0.7	**9.28 ± 0.5**
Frogs	24.0 ± 8.5	16.0 ± 7.3	8.0 ± 5.4	20.0 ± 8.0	-0.88 ± 0.3	-1.04 ± 0.2	-1.2 ± 0.2	-0.68 ± 0.2
Missile Command	20.0 ± 8.0	20.0 ± 8.0	20.0 ± 8.0	**56.0 ± 9.9**	-1.44 ± 0.3	-1.44 ± 0.3	-1.28 ± 0.3	**3.24 ± 1.3**
Portals	12.0 ± 6.5	**28.0 ± 9.0**	16.0 ± 7.3	**28.0 ± 9.0**	0.12 ± 0.06	**0.28 ± 0.09**	0.16 ± 0.07	**0.28 ± 0.09**
Sokoban	0.0 ± 0.0	0.0 ± 0.0	**4.0 ± 3.9**	**8.0 ± 5.4**	0.16 ± 0.1	0.32 ± 0.1	**0.7 ± 0.2**	**0.6 ± 0.1**
Survive Zombies	44.0 ± 9.9	36.0 ± 9.6	52.0 ± 10.0	44.0 ± 9.9	13.28 ± 2.3	14.32 ± 2.4	18.56 ± 3.1	21.36 ± 3.3
Zelda	8.0 ± 5.4	**20.0 ± 8.0**	8.0 ± 5.4	**28.0 ± 9.0**	0.08 ± 0.3	0.6 ± 0.3	0.8 ± 0.3	0.6 ± 0.3
Overall	21.6 ± 2.6	23.6 ± 2.7	22.0 ± 2.6	**49.2 ± 3.2**	9.0 ± 0.9	10.0 ± 1.0	10.7 ± 1.0	**13.5 ± 1.2**

TABLE II: Percentage of victories and scores from each game. The results in bold are the best from each game. Each value corresponds to the average result obtained by playing that particular game 25 times.

the action that moves the avatar into the first lane will most likely cause the player to lose the game. A typical behaviour observed in this game is the agent moving parallel to the road without ever crossing it, trying to find gaps to cross, but too "scared" to actually try it.

VII. Conclusions

This paper explored the performance and problems of a vanilla Monte Carlo Tree Search (MCTS) algorithm in the field of General Video Game Playing (GVGP). Several modifications to the algorithm have been implemented in order to overcome the problems, such as rewarding the discovery of new sprites, augmenting the knowledge of other elements in the game, and using past experience to ultimately guide the MCTS roll-outs. Results show a significant improvement in performance, both in percentage of victories and scores achieved. These improvements work better in some games than in others, and reasons for this have also been suggested.

This work presages multiple future extensions, such as the implementation of a general path-finding algorithm for better distance measurements, or experimenting with different algorithms to bias roll-outs. This study features a $(1+1)$ Evolution Strategy to guide the Monte Carlo simulations, but more involved evolutionary techniques will be explored, as well as other approaches like gradient descent methods.

GVGP, with the absence of game-dependent heuristics, has proven to be a challenging and fascinating problem. It reproduces current open challenges in Reinforcement Learning, such as the absence of meaningful rewards (as explained for *Frogs*, in Section VI). We hope that this research, and also the new General Video Game Competition [15], helps to shed some light on a problem that is still unresolved, and also to bring more researchers to this topic.

Acknowledgment

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

References

[1] Atif Alhejali and Simon M. Lucas. Using Genetic Programming to Evolve Heuristics for a Monte Carlo Tree Search Ms Pac-Man Agent. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 65–72, 2013.

[2] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[3] Amit Benbassat and Moshe Sipper. EvoMCTS: Enhancing MCTS-based Players Through Genetic Programming. In *Proceedings of the Conference on Computational Intelligence and Games (CIG)*, pages 57–64, 2013.

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

[5] Hilmar Finnsson and Yngvi Björnsson. Simulation-based Approach to General Game Playing. In *Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 259–264, 2008.

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

[7] Marc Gendron-Bellemare, Joel Veness, and Michael Bowling. Investigating Contingency Awareness using Atari 2600 Games. In *Proceedings of the Twenty-Sixth Conference on Artificial Intelligence (AAAI)*, pages 864–871, 2012.

[8] Michael Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI Competition. *AI Magazine*, 26:62–72, 2005.

[9] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A Neuroevolution Approach to General Atari Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, DOI:10.1109/TCIAIG.2013.2294713:1–18, 2013.

[10] John Levine, Clare B. Congdon, Michal Bída, Marc Ebner, Graham Kendall, Simon Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General Video Game Playing. *Dagstuhl Follow-up*, 6:1–7, 2013.

[11] Simon M. Lucas, Spyridon Samothrakis, and Diego Perez. Fast Evolutionary Adaptation for Monte Carlo Tree Search. In *Proceedings of EvoGames*, page to appear, 2014.

[12] Jean Méhat and Tristan Cazenave. A Parallel General Game Player. *KI - Knstliche Intelligenz*, 25:43–47, 2011.

[13] Maximilian M¨ller, Marius Thomas Schneider, Martin Wegner, and Torsten Schaub. Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI - Knstliche Intelligenz*, 25:17–24, 2011.

[14] Yavar Naddaf. Game-Independent AI Agents for Playing Atari 2600 Console Games. Master's thesis, University of Alberta, 2010.

[15] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon Lucas. The General Video Game AI Competition, 2014. www.gvgai.net.

[16] Tom 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, 2013.

[17] Shiven Sharma, Ziad Kobti, and Scott Goodwin. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *Proceedings of the 21st Australasian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pages 49–55, 2008.