Utrecht University

Analysing And Improving The Knowledge-based Fast Evolutionary MCTS Algorithm

Master Thesis by

Jim van Eeden, 3344800

Supervisor:

dr. ir. D. Thierens

MSc. Technical Artificial Intelligence
Faculty of Science
Department of Information and Computing Science

July 2015

Utrecht University

Abstract

MSc. Technical Artificial Intelligence
Faculty of Science
Department of Information and Computing Science

by Jim van Eeden, 3344800

In the ongoing strive to reach human-level intelligence with intelligent agents, it has become clear that the agents should be able to deal with a host of different problem domains. Being very good on one particular domain, like chess for instance, does not suffice. That is why the area of General Video Game Playing (GVGP) was introduced. Here, agents will play a wide range of different video games, while they do not know which game they will be playing beforehand. The agents are forced to learn the game by playing. This master thesis will research one particular version of the Monte Carlo Tree Search algorithm that was created for GVGP, called the Knowledge-based Fast Evolutionary (KB Fast-Evo) MCTS algorithm. This algorithm uses a knowledge base to store information about the game and also an evolutionary approach to bias game simulations. First a general study of the MCTS algorithm is given, together with a detailed explanation of General Video Game Playing. Then the KB Fast-Evo algorithm is analyzed, where a lot of room for improvement becomes apparent. In the final chapters an attempt is made to improve the KB Fast-Evo algorithm, by changing the evolutionary approach, by adding path finding and some other fine-tuning. The final algorithm does perform a lot better, but there is still room for improvement. However, it is not likely the algorithm will ever be optimal on its own.

Acknowledgements

I wish to express my sincere thanks to Starapple for giving me a workplace, coffee and other facilities during the writing of this thesis.
I am also grateful to Dirk Thierens, my thesis supervisor, for sharing his expertise and his guidance throughout the process.
Finally, I wish to thank the creators of the GVGAI Competition (gvgai.net) for making such an interesting platform for AI research.

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) is one of the most exiting and most talked about fields of study in Computer Science. A lot of research has been dedicated to this field, where the ultimate goal would be creating an agent with human-level intelligence. However, even with the fast advances in technology over the past few decades, we've only seen the tip of the iceberg when it comes to the potential of AI. This thesis will try and constitute some useful research in order to make advances in the field.

1.1 The Approach

The particular algorithm that was chosen to be researched is a variant of the new and promising Monte Carlo Tree Search algorithm (MCTS). This algorithm uses a lot of simulations in order to make a fair estimate of what actions the agent should take to reach its goals. The MCTS algorithm has seen some great success, mostly in (board) games, such as Go for instance. But can an agent be deemed intelligent if it is only good in playing one particular game? Most researchers agree that an intelligent agent should be able to solve all different kinds of problems. That is why General Video Game Playing (GVGP) was introduced. In this particular field in AI research, agents have to play different kinds of video games, without them knowing the rules beforehand. So an agent is forced to learn the games while playing them. In this thesis the Knowledge-based Fast Evolutionary (KB Fast-Evo) MCTS algorithm will be analyzed and improved, with the goal of creating a better General Video Game Playing agent. All of the above will be extensively covered in the following chapters of this thesis.

1.2 Research Questions

In order to give structure to the research in this thesis, some research questions have been formulated. The answers will be covered in the chapters to follow and the research questions will be revisited and discussed in the final chapter.

- How does the KB Fast-Evo algorithm compare to a standard MCTS implementation for GVGP?
- What improvements can be made to the KB Fast-Evo algorithm for GVGP?
- What other techniques can be used to maximize performance of the MCTS algorithm for GVGP?

1.3 Thesis Overview

The first chapters will give an overview and detailed explanation of the topics covered. Chapter 2 *"Monte Carlo Tree Search"* is a literature study of the MCTS algorithm. The MCTS algorithm is explained here and the most important and relevant research about the algorithm is also mentioned. Chapter 3 *"General Video Game Playing"* describes GVGP in detail and also the platform on which the research was conducted is given here.

The next chapters cover the actual research, where the research questions are answered. In Chapter 4 *"Implementation Analysis"* the KB Fast-Evo algorithm is implemented on the GVGP platform and compared to the normal MCTS algorithm. Then it's analyzed in detail and some issues will become apparent that hurt the performance of the algorithm. Chapter 5 *"Improving KB-Fast Evo"* will then try to solve the issues with the KB Fast-Evo algorithm that were previously found. Additional interesting experiments and an attempt to maximize performance of the KB Fast-Evo algorithm are shown in Chapter 6 *"More Improvements And Experiments"*.

In the final Chapter 7 *"Discussion And Conclusion"*, a discussion about the research can be found, together with thoughts about what can be learned from it. The research questions are also revisited and answered in short. Finally, suggestions for future work are given there.

Chapter 2

Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm is a relatively new algorithm that has received a lot of scientific interest in the past few years. It has had some promising results in the area of game playing and decision making. Games such as Go and Hex, where human players used to be far ahead of computers, is where the MCTS algorithm has shown to be very successful. But also in the area of General Game Playing (GGP) and General Video Game Playing (GVGP), where the rules of a game aren't given beforehand, it has become one of the most promising algorithms.

2.1 Background

The MCTS algorithm is based on earlier ideas of random sampling (1940s) and tree search (1928), but the name Monte Carlo Tree Search was first coined in 2006, where R. Coulom used it to play the game Go [1]. One very popular variant of the MCTS algorithm is Upper Confidence Bounds for Trees (UCT). This version was developed by L. Kocsis and C. Szepesv [2] and uses the idea of Upper Confidence Bounds (UCB).

The MCTS algorithm is mostly applied to *game-theoretic* domains, as is also the case in this thesis. Those domains are games with a set of established rules in which one or more players (agents) act strategically to maximize personal utility. In other words, the players try to win the game and/or get a high score. More formally, a game may be described by the following components:

- S: The set of states, where s_0 is the initial state.
- $S_T \subseteq S$: The set of terminal states.
- $n \in \mathbb{N}$: The number of players.

- A: The set of actions.
- $f : S \times A \rightarrow S$: The state transition functions.
- $R : S \rightarrow \mathbb{R}^k$: The utility function.
- $\rho : S \rightarrow (0, 1, \ldots, n)$: Player about to act in each state.

A game starts in state S_0 and progresses over time $t = 1, 2, \ldots$ until some terminal state S_T has been reached. A player k_i takes an action $a \in A$ that leads via f, to the next state s_{t+1}. Each player receives a reward for their performance defined by R. Usually in board games, non-terminal states have a reward of 0 and terminal states have a reward of +1, 0 or -1 for a win, draw, or loss, respectively. Video games typically have a more intricate scoring system that awards (negative) points for non-terminal states. The *strategy* of each player determines the probability of selecting action a in state s. The player's strategy should ideally be such that an optimal action is selected every time, but this is a difficult task of course. This thesis aims to get closer to that ideal performance in the area of General Video Game Playing.

2.2 The MCTS Algorithm

The vanilla version, i.e. basic unmodified version of the Monte Carlo Tree Search algorithm will be explained in this section. The idea is to continually run simulations of an entire game from its current state and building a game tree in the memory. The tree will contain an approximation of the true value for all the possible actions from the current state and further game states. When the deliberation time is up, the algorithm selects an action based on the values it has given to each action. Then, when another game state is reached and a new action has to be picked, the algorithm starts over and will build a new tree again.

Figure 2.1 shows one iteration of the MCTS algorithm. The algorithm will continue to build the tree until a predefined computational budget has been reached. This is usually a time constraint, memory constraint or a limit to the amount of iterations. A node in the tree represents a state in the game, with the root as the current actual state in the game. A child node is a state that is reached by performing an available action from the state represented by the parent node. As can be seen from the figure, an iteration is comprised of four steps.

1. Selection

Starting at the root node, the algorithm will descend down the tree using some

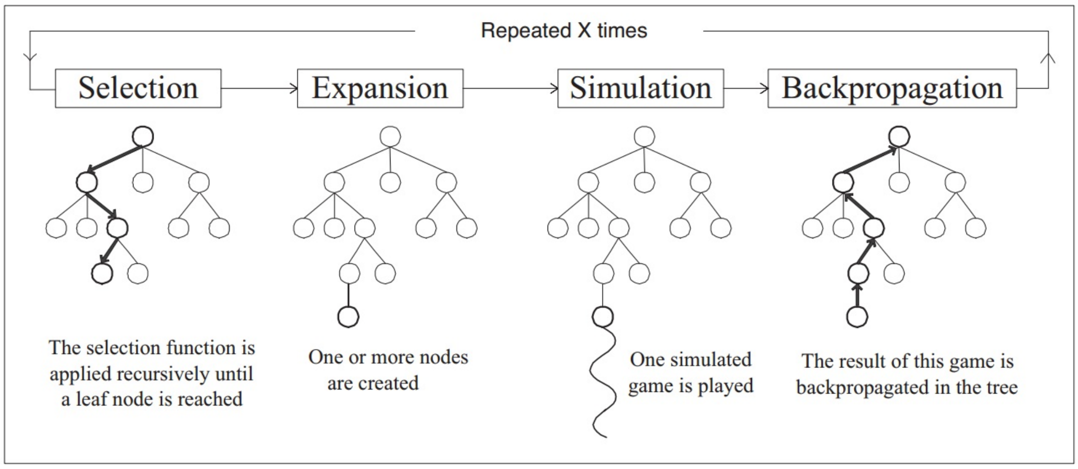
Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

FIGURE 2.1: This figure shows the basic steps of one iteration of the MCTS algorithm. Each time an action has to be picked these steps are repeated, until the stopping condition is met. Image from [3].

child selection policy until an expandable node is reached. A node is expandable if a new child node can be added to that node, i.e. the node does not represent a terminal state of the game and there are reachable states that have not yet been added to the tree.

2. **Expansion**

   One (or more) child nodes are selected and added to the tree, according to the reachable states from the current node.

3. **Simulation**

   From the added node's state, the game is played out to the end according to the *default policy*. In the vanilla MCTS algorithm this is simply selecting actions at random.

4. **Backpropagation**

   The result of the simulation is backed up through the tree by updating the win/loss ratio in all the nodes that were traversed during the selection and expansion step.

When the computational budget is reached, one of the children of the root node will be selected as the next action to be played. This selection usually is either the child node with the highest win/loss ratio, the child node with the most visits, or a combination of those two.

The steps above may be grouped into two distinct policies.

1. **Tree policy**

   Select or create nodes in the current tree. (Selection and expansion)

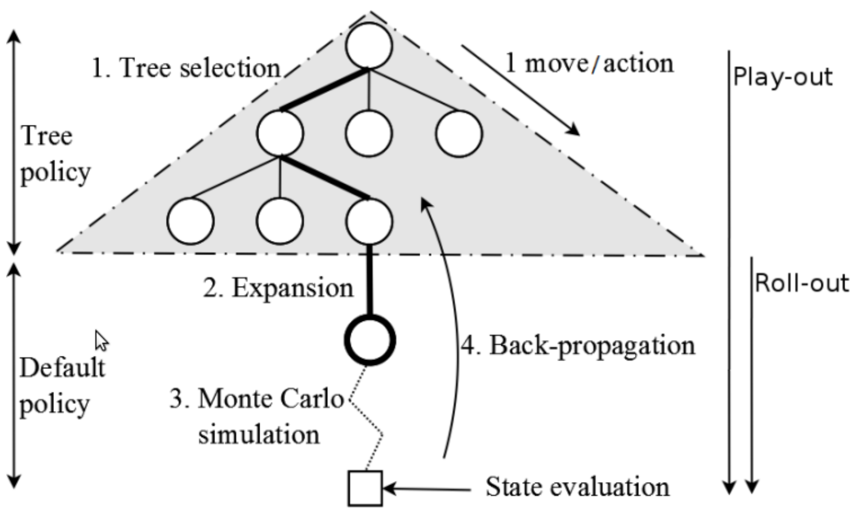

FIGURE 2.2: The MCTS Algorithm in one overview. Image from [4].

2. Default policy

Play out the entire game in some way from a given state. (Simulation)

The backpropagation step does not use a policy itself, but does influence future tree policy decisions. For the tree policy often UCB1 is applied as proposed by Kocsis and Szepesvári [2]. This is referred to as Plain UCT, and balances the selection of child nodes out nicely by appropriately choosing between nodes that seem to lead to good results (exploitation) and nodes that haven't been visited much, but might be rewarding in the future (exploration). The UCT version of the MCTS algorithm will be explained more in detail in section 2.4. In vanilla MCTS the default policy plays out the game by selecting next actions at random. This might seem like a bad way to go, but it is very fast and therefore leads to a lot of samples, which eventually does lead to informed decisions. So even the non-enhanced vanilla MCTS works decently. More on enhancements later in section 2.5.

2.3 Characteristics

This section will shortly explain some of the characteristics of the MCTS algorithm. First of all, the algorithm is *aheuristic*. This means there is no need for any domain specific knowledge, such as an evaluation function, for the algorithm to perform well. The full-depth minimax algorithm is of course the very best you can hope for in a game-theoretic sense, because you have full knowledge of how each move can play out. But for most games a full-depth minimax tree would simply be too large to work with. A limited-depth minimax algorithm could be used, but then you would need a good evaluation function for the leaf nodes. This is why the MCTS algorithm is better in some domains; sometimes a good evaluation function just doesn't exist (yet). Moreover, it has been proven that given an infinite number of simulations, the UCT algorithm with a default policy with random roll-out will eventually yield the correct full-depth minimax tree and is thus optimal [5].

Furthermore, the MCTS algorithm is an anytime algorithm, meaning that it can be stopped at any point during run-time and it will still return a reasonably good answer. Of course running more simulations will yield a better result, as running infinitely will actually result in the optimal answer, as stated above. This property is one of the reasons why it is applied in General Video Game Playing, as actions are required at run-time and the MCTS algorithm can take maximum advantage of the small amount of time between each action.

Another characteristic of the MCTS algorithm is that it grows asymmetric search trees. In particular the UCT algorithm focuses more towards the most promising nodes, without the selection probabilities of other nodes ever reaching zero. Over time this leads to an asymmetric tree, which can be used more efficiently than a symmetric tree. The shape of the tree could even be used to analyze the game itself, which has been done by Williams [6] for instance.

2.4 The UCT Algorithm

The Upper Confidence Bounds for Trees or UCT algorithm was already mentioned, but will be explained a bit more in detail in this section. UCT is a variant of the MCTS algorithm that uses Upper Confidence Bounds (UCB) for the selection policy. Kocsis and Szepesvári proposed to use UCB1 for this in 2006 [2]. UCB1 is the simplest version of UCB and was proposed by Auer et al. in 2002 [7]. This version is so popular because it is simple, sound, and offers a nice way to balance the exploitation versus exploration trade-off.

The formula used is the following and returns what action should be picked next from a given state:

$$a^* = argmax_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\tfrac{2 \ln N(s)}{N(s,a)}} \right\} \tag{2.1}$$

$Q(s,a)$ is the average return of a state-action pair, which is a value [0..1]. Simpler said, this is how well action a has been doing when the game was played out (simulation step) from state s and that action was picked first. $N(s)$ is the number of times the algorithm has visited state s and $N(s,a)$ is the amount of times action a has been sampled from state s. $A(s)$ is the set of possible actions from state s. Finally, C is a constant $\geqslant 0$.

To ensure each child is sampled at least once before any sibling is expanded further, the formula will return ∞ when $N(s,a) = 0$. If a state has more children that aren't visited yet, one of the children will be selected at random.

The term added to $Q(s,a)$ is called the UCT bonus. The UCT bonus is the exploration part of the formula, where $Q(s,a)$ itself is the exploitation part. Every time an action is selected, its UCT bonus value goes down, as $N(s,a)$ is incremented. Meanwhile their siblings' bonus value goes up, because $N(s)$ is incremented. So when 'good' actions have been sampled enough times, their siblings have had their bonus increase enough to start exploring those as well.

The constant C can be adjusted to lower or increase the amount of exploration. Usually the value is chosen experimentally, but some researchers have tried to tune it automatically too. Kocsis and Szepesvári have shown that $C = 1/\sqrt{2}$ satisfies the Hoeffding inequality for rewards that lie in range [0..1] [5]. In research papers, often the $C = \sqrt{2}$ seems to be used.

2.5 Existing Improvements

Since MCTS was first proposed, a lot of research has been dedicated to improving the algorithm, either in general or for a specific application. The most important and relevant improvements for the MCTS algorithm are mentioned in this section. The enhancements that are shown below have been proposed by different researchers in the past few years and improve upon MCTS by changing or adding to different parts of the algorithm.

2.5.1 UCB1-Tuned

UCB1-Tuned was proposed by Auer et al. themselves as an enhancement to their UCB1 [7]. They found in their own experiments that it performed better than the original algorithm. UCB1-Tuned is commonly used in MCTS implementations as an improvement to UCB1, including Go [8], Othello [9], and the real-time game Tron [10]. Although not all mentioned papers found that UCB1-Tuned improved their algorithm as compared to normal UCB1.

In UCB1-Tuned the UCB1 formula is changed by replacing the upper confidence bound. The whole formula is the following:

$$a^* = argmax_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\tfrac{\ln N(s)}{N(s,a)} \min(\tfrac{1}{4}, V(s,a))} \right\} \tag{2.2}$$

where

$$V(s,a) = \left(\frac{1}{N(s,a)} \sum_{t=1}^{N(s,a)} Q_t(s,a)^2 \right) - \left(\frac{1}{N(s,a)} \sum_{t=1}^{N(s,a)} Q_t(s,a) \right)^2 + \sqrt{\tfrac{2 \ln N(s)}{N(s,a)}}$$

Notice that $\frac{2 \ln N(s)}{N(s,a)}$ from the UCB1 formula is replaced by $\frac{\ln N(s)}{N(s,a)} \min(\frac{1}{4}, V(s,a))$. The left hand side of formula $V(s,a)$, before the plus sign, is the sample variance of action a in state s. $Q_t(s,a)$ is the result of play-out number t from state s when action a was selected. The factor $\frac{1}{4}$ is an upper bound on the variance of a Bernoulli random variable. It should be noted that Auer et al. were not able to prove a regret bound for UCB1-Tuned, but experimentally found that it performed better.

2.5.2 Decisive and Anti-Decisive Moves

This is a simple technique that checks for each action in the selection and simulation steps if it is decisive or anti-decisive. A decisive move is an action that will win the game. An anti-decisive move is an action that prevents the opponent from making a decisive move. If the action is found to be either decisive or anti-decisive, it will always be picked. This was proposed by Teytaud and Teytaud [11] and they show that this modification significantly increases playing strength, even when the increased computational cost of checking for decisive moves is taken into account.

2.5.3 Transpositions

Different sequences of moves in games could often lead to the same state. Two such sequences would be two different paths in the MCTS tree. But the underlying game could also be represented by a Directed Acyclic Graph (DAG), and in this case the two sequences that lead to the same state could be represented by a single edge in the DAG. Whenever two nodes in the MCTS tree represent the same game state, this is referred to as a transposition.

Using transpositions, extra information can be extracted from the simulations and stored in the nodes of the transposition. All transpositions are usually stored in a transposition table that keeps track of which nodes are in the same transposition. Brodeur et al. [12] and Kozelek [13] both use transpositions in their algorithm and find that it improves upon the plain UCT algorithm. Both of them have an adapted score value for nodes that takes the transpositions into account. All the nodes in the same transposition share the same score value. This is done by propagating a score update of a node to all nodes in the same transposition using the transposition table. Nodes in the same transposition do not share the same visits, as this could lead to a misleading bias as explained in [12].

2.5.4 Parallelisation

Parallelisation is an enhancement that exploits the independent nature of the simulations in MCTS. Multiple simulations can be running at the same time and this makes for a more efficient algorithm, especially when a computer can make use of multiple cores. But this also raises some issues, such as combining the results of the simulations in the right way, and synchronising the threads. There are several ways to deal with this and they will be described shortly here.

Leaf parallelisation
: Runs multiple simulations from one leaf. This is supposed to improve the initial estimate of the leaf and thus influence the selection and expansion steps in a positive way. One problem is that the algorithm has to wait for the longest simulation to finish before continuing.

Root parallelisation
: Also named multi-tree MCTS. This builds multiple trees at the same time, so it just runs the same algorithm in different threads. The advantage is that the anytime property is still maintained. At the end a majority voting scheme is used to select the action. Soejima et al. show that this technique improves upon non-parallelised MCTS [14].

Tree parallelisation
: Runs simultaneous simulations on different parts of the same tree. The issue here is that the different threads should not be able to access the same areas of the tree at the same time. This is done by locking either the whole tree when a thread uses it for node updates (global lock), or by locking each individual node (local lock). Also because different threads will often follow the same path down the tree, a virtual loss is applied to a node when the first thread reaches it. This will push the other threads to explore other areas of the tree. Bourki et al. find that this performs slightly better than root parallelisation [15].

2.5.5 History Heuristic

This improvement is closely related to the history heuristic in Alpha-Beta search [16]. The idea is to store information in some way that can be used again to influence decisions later on; in the selection or simulation step in the MCTS case. Gelly and Silver used a Grandfather heuristic [17], which initializes a node with the value of it's grandfather. This assumes a move is still good after the opponent has played a move. Gelly and Silver found that this did not improve the algorithm as much as other initialization methods.

Kozelek [13] collected statistics (average value, number of visits) on every step in the tree, regardless of the level. This was then used to influence the selection step by including a value derived from these statistics in the UCB formula, using a technique from Chaslot et al. called Progressive Bias [18]. He found that this technique had a very positive impact on the performance of the algorithm.

The following three subsections also show history heuristics, but these techniques are mainly used in the simulation step. They were introduced by Björnsson and Finnsson for their world champion General Game Playing agent CadiaPlayer [19], [20]. They show that these techniques all improve on the vanilla MCTS algorithm, but none dominates the others. Some of the schemes work better for certain games, as the properties of different games favor different techniques.

2.5.6 MAST

The Move-Average Sampling Technique or MAST was introduced by Björnsson and Finnsson in [19]. For each action a in a full play-out, the average return of that play-out is stored $Q_h(a)$. This value is then used to bias which unexplored action to investigate next, both in the simulation step and in the expansion step for unexplored child nodes. This is done using Gibbs sampling, with the formula below.

$$P(a) = \frac{e^{Q_h(a)/\tau}}{\sum_{b=1}^{n} e^{Q_h(b)/\tau}} \tag{2.3}$$

$P(a)$ is the chance action a will be chosen in the current state. Actions with a higher $Q_h(a)$ are more likely to be chosen. τ is a constant that stretches or flattens the distribution. $\tau \to 0$ stretches the distribution, whereas higher values make it more uniform.

There is a slight variation of this called Tree-Only MAST (TO-MAST), which uses only the actions that are in the search tree (so not the actions in the simulation), to update the $Q_h(a)$ value. This leads to less samples for the value, but generally the actions in the tree are more informed, so the quality of the samples should be better.

2.5.7 PAST

Predicate-Average Sampling Technique or PAST works as MAST, but uses predicates to get a finer generalisation than MAST. Every state has certain predicates that are true in that state. Now instead of a value $Q_h(a)$, a value $Q_p(p, a)$ is taken. $Q_p(p, a)$ is also updated in the back propagation step, but now all values for action a in combination with all $p \in P(s)$ are updated. $P(s)$ are all true predicates in state s. An action is chosen again with Gibbs sampling, with the maximum value of $Q_p(p, a)$ over all predicates p in the current state.

2.5.8 FAST

Feature-Average Sampling Technique (FAST) makes use of game features. Understanding game features is essential for high level game playing, such as in chess, where knowing the different piece types is important. Or for example in Othello, where the different squares have varying importance (corners are high value). In FAST, the different features are extracted from the game description and then the relative importance is determined using $TD(\lambda)$ [21]. The feature values are then used to calculate a value $Q(a)$, which is used for action selection as in MAST and PAST. Different game types have different ways of calculating $Q(a)$. In games such as chess, taking high ranking pieces with lower ranking pieces will result in a higher value. In games such as Othello, having your piece on an important square will result in a higher value.

2.5.9 AMAF

This technique is closely related to history heuristics. AMAF stands for All Moves As First and was first proposed to use in UCT for the game Go by Gelly et al. [17]. It was proven to be very successful in Go [22]. There are many variations to this enhancement, but the basic idea is to treat each move during selection and simulation as if they were the first move to be played. It follows the assumption that if a move was good when played first, it is good in general. This means that when the play-out passes through a node, that node will be updated *and* all its siblings that represent moves encountered further in the play-out. This is because the same outcome would (possibly) be reached when the order of the moves had been different. An example is shown in Figure 2.3.

Some implementations keep a separate AMAF score together with the normal score instead of combining them. A few different AMAF enhancements are described below.

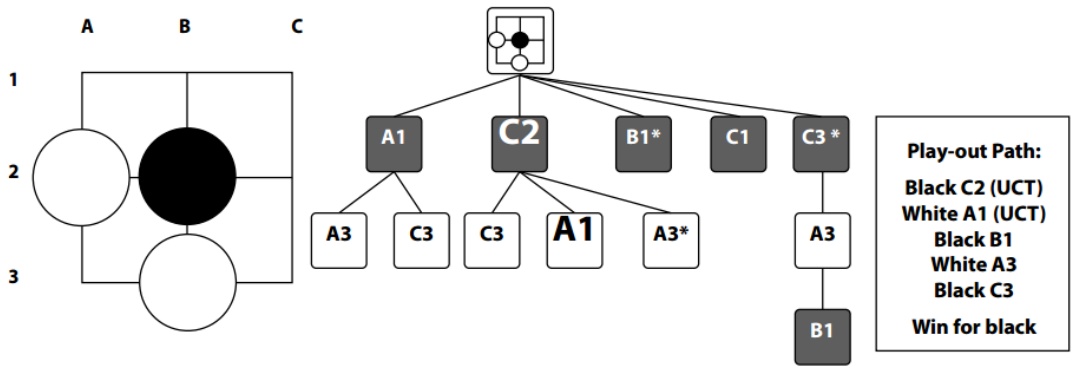

FIGURE 2.3: This figure shows a simple 3x3 Go game, where the MCTS algorithm has already built a partial tree. The next play-out can be seen on the right. UCT selects Black C2 and White A1, then the simulation plays Black B1, White A3 and Black C3. Black C2 was played first here, but Black B1 or C3 could also have been played first to get the same final board. So all these nodes have their score/visits updated. The same goes for White A1 and A3. All nodes labelled with an asterisk (*) get the extra AMAF update. Image from [23].

α-AMAF

Keeps a separate AMAF score and combines it with the normal UCT score using an α value. This was proposed Helmbold and Parker-Wood in [23]. The formula looks as follows:

$$a^* = \alpha A + (1 - \alpha)U \tag{2.4}$$

where A is the AMAF score and U the UCT score.

RAVE

Was also proposed in [23]. This has become a very popular enhancement in computer Go agents such as the well known MoGo. It is similar to α-AMAF, except that the α value used at each node decreases with each visit. The following formula is used to calculate the α value:

$$\alpha = \max\left\{0, \frac{V - N(s)}{V}\right\} \tag{2.5}$$

Where V is a positive integer representing the amount of visits a node will have before the RAVE value isn't used at all anymore. $N(s)$ is the amount of visits for a node.

There are a few more AMAF versions such as Killer-RAVE, which only uses the RAVE update for the most important moves. Also PoolRAVE, which builds a pool from the best moves according to their RAVE values, then selects one move and plays it with probability p and otherwise plays the default policy. There are even more varieties, but it is noteworthy to mention the conclusion of HelmBold and Parker-Wood [23] when they compare the AMAF variants:

- Random roll-outs provide more evidence about the goodness of moves made earlier in the roll-out than moves made later.
- AMAF updates are not just a way to quickly initialise counts, they are useful after every play-out.
- Updates even more aggressive than AMAF can be even more beneficial.
- Combined heuristics can be more powerful than individual heuristics.

2.5.10 Evolutionary Learning

This is a technique that was already applied to other algorithms and used to solve different problems. It's also known as genetic programming. In the case of MCTS, the aim is to evolve some policy that biases the roll-out so that more reasonable actions are picked, while trying to keep the decision making as fast as possible. The latter of course is a returning aspect of MCTS, because more play-outs will lead to a better result. The policies are evolved offline, meaning that the players in the population will play against some benchmark player. All the players will be evaluated, then mutation, crossover and selection will be applied to the population which should result in an overall stronger population. After a set number of iterations, the player with the highest fitness will be selected and that policy will be used. For more information on genetic programming in general, see [24]. This technique can be applied in many forms and following are a few examples of research in this area. One of the first people who applied genetic programming to MCTS was Cazenave [25]. He used it to evolve a heuristic function for the default policy in the game Go and found that it outperformed UCT with RAVE. Pettit and Helmbold [26] evolved board patterns with weights for the game Hex that also influences action selection in the simulation step. This version had a 90% win rate against vanilla MCTS. Benbassat and Sipper [27] called their method EvoMCTS. They evolved evaluation functions for action selection. Their technique is applicable to many games. Finally Alhejali and Lucas [28] apply genetic programming to a MCTS Ms Pac-Man agent. They also try to evolve a good evaluation function for the simulation step.

It must be noted that the following two techniques were applied to real-time games. This means that the MCTS algorithm doesn't have as much time to run (~40ms) as it would normally have with a board game (several seconds). Given the small amount of running *time, the amount of steps in a roll-out is set to a fixed limit, instead of completely playing out a game.*

2.5.11 Fast Evolutionary MCTS

Fast Evolutionary MCTS was proposed by Lucas et al. [29]. It also uses evolutionary learning, but differs from all other research in that it evolves a heuristic online, rather than offline. The previous technique of evolutionary learning would play an entire game, after which the evolution to the population is applied. Fast Evolutionary MCTS evolves a weight matrix that biases the default policy *during* the game, and applies evolution after a number simulations. The algorithm starts with a feature set (each game state has different feature values, see FAST in subsection 2.5.8) that is hand picked and a weight matrix that is evolved. The weight matrix contains a weight for each feature in combination with each action. Note that there are usually a lot less actions in video games than board games, so the matrix will be relatively small. Using the feature set and the weight matrix, each action is given a value, and like in MAST (subsection 2.5.6), Gibbs sampling is used for action selection.

To evolve the weight matrix a (1+1) evolutionary strategy is used. This is a very simple evolutionary strategy. First the weight matrix is mutated, then a set number K of play-outs is ran. After that, the mean of the K play-out outcomes is taken and set as the fitness of the current weight matrix. Because the roll-outs are a fixed depth, the game is usually not played until the end. So the outcome of the play-out is some kind of game score. The matrix with the highest fitness value in the population is chosen each time and mutated again, the lowest is dropped from the population. So the population only has a size of two.

Lucas et al. applied this algorithm to the Mountain Car problem and a simplified version of Space Invaders. They conclude that the algorithm works a lot better than the vanilla MCTS algorithm, but it doesn't perform well every single game. There are some games in which the algorithm isn't able to evolve a good weight matrix. They also tested with a pre-evolved weight matrix, which was the weight matrix that resulted after the algorithm played a game particularly well. This lead to high scores in the games each time.

2.5.12 Knowledge-Based Fast Evolutionary MCTS

This is also referred to as KB Fast-Evo and is an extension of the previous technique. It was also proposed by Lucas et al. [4]. It is focused towards General Video Game

Playing. There are small changes in the Fast Evolutionary technique, but the main benefits come from combining the evolutionary strategy with a knowledge base. First of all, features are no longer hand picked, but gotten from the game description. These could be NPCs, resources, portals, etc. depending on the game. This means that there is no longer a fixed amount of features, because the amount of features can change during the game. So the size of the weight matrix can now change. The feature values are the closest distance to each feature of that type. Secondly, evolution is now applied after each simulation. This means a different fitness evaluation is needed. The scoring function is now used both for updating node values *and* fitness evaluation at the end of each play-out.

For each event i.e. collision with a sprite during a play-out, two values are kept: the number of occurrences and the average score change. These values are then used at the end of each play-out to calculate three other values:

- Score change. Simply the difference in score between the beginning and end of the play-out.
- Knowledge change. This is a value calculated that will be higher when a play-out produces more events, especially when those events have rarely occurred.
- Distance change. Will be a higher value when the avatar has reduced its distance to sprites that are unknown or that generated high-reward events in the past.

The first value, score change, is used as the score/fitness value at the end of the play-out. But if it is 0, a combination of the second and third value is used.

The knowledge base consists of a combination of two factors, namely *curiosity* and *experience*. Discovering new events is the curiosity part. Both the second and the third value steer towards this. Getting a score boost (from events that lead to a score boost in the past) is the experience part; the first and third value guide actions towards this factor.

This technique was tested on a General Video Game Playing platform (more on this in section 3.2). The separate Fast Evolutionary technique and the Knowledge Base technique were also tested. Both of these did not perform much better than the vanilla MCTS algorithm in the different video games. However, the combination did perform a lot better on most of the games, leading is some cases to a 100% victory rate. But for other games there was nearly no improvement. Lucas et al. found that some of the issues were due to the fact that the distance calculation didn't take obstacles into account, which was important for certain games. Moreover, sometimes features weren't specific enough, for instance when the direction of a collision is important. Finally, the scoring system of some games would make it hard for the algorithm to find the goal of the game. For instance in the game Frogger, where it is really easy to die crossing a road, but reaching the other side of the road is the only time a reward (score increase) is given.

Chapter 3

General Video Game Playing

The idea to benchmark Artificial Intelligence through games has been around since AI itself. In the early days, games such as Chess were used to do this, as was also proposed by Turing. But since *Deep Blue* beat the reigning world champion Gary Kasparov in a game of chess, largely using domain-specific heuristics, the benchmark was re-evaluated. If an agent was really good at one specific thing, could it be considered intelligent? The general opinion was: No, because different games require different cognitive skills. So in 2005, the field of General Game Playing (GGP) was introduced, where the agent does *not* know beforehand what game he will be playing. A yearly GGP competition is still hosted [30]. This has resulted in quite some research papers on the topic, often with some good improvements on existing research or new ideas in the field of artificial intelligence. In the previous chapter some of these were already mentioned, like Björnsson and Finnsson [19] for example.

In GGP usually turn-taking board games are played. The agents do not know beforehand what games they will be playing; only at the start of the game the rules will be specified using GDL, the Game Description Language [31]. A turn-taking board game is often quite slow-paced. Just like human players, agents get seconds or even minutes to do an action. However, in the ongoing strive for reaching human-level intelligence, it is clear that a different kind of general intelligence is also desired. Humans often face problems that are quite different from board games, not in the least in the amount of time they have to make a decision.

In 2013 the new field of General Video Game Playing (GVGP) was proposed in order to broaden the perspective on General Artificial Intelligence [32]. GVGP is based on the same idea as GGP, namely that one agent has to excel at multiple games, without knowing the games in advance. But GVGP is based on video games, rather than board games. This leads to some important differences in the problems the agents face. They have to react real-time, usually within a time frame of about 40ms. The environments are also much more unpredictable, noisy and dynamic, compared to games in GGP. Also the current focus is on single-player games, similar to the old Atari games, where GGP games are usually multi-player. Lastly, the rules of the game are not given in GVGP, where in GGP they are. The first General Video Game AI competition was held at the IEEE Conference on Computational Intelligence and Games (CIG) in 2014 [33], and a new one will be hosted in 2015 as well.

3.1 Prospects And Pitfalls

This section will give a short description of the prospects and pitfalls of General Video Game Playing, in order to give a fair idea of the area.

As mentioned above, one of the most important prospects of GVGP is setting a benchmark for human-level intelligence. Previously this was done with specific games or the Turing test. Lehman and Miikkulainen propose replacing those with GVGP [34], as GVGP will be able to test many different levels and aspects of human intelligence. Also it would be easy to scale up in difficulty (of the games) as AI capabilities mature. Furthermore, agents will no longer be able to fit game-specific heuristics to problems (unless they learn them by playing), so the agents could more easily be considered to have 'real' AI. Lastly, one could say that video games are already a big part the human problem domain, so it would be a logical area to explore.

There are also some aspects to GVGP that should be regarded carefully, before going further into it. Firstly, agents will be very dependant on the game's scoring system. Some games will be designed (balanced) better than others, so games that are used should be selected with care. Moreover, for a correct benchmark of human-level AI, the inputs and outputs for agents and humans should be the same. So agents would get visual and auditory inputs, and controller-type outputs, just like humans. Currently this is not the case (yet) for the GVGAI Competition [33]. More on the framework that is used for that competition in section 3.2. Another thing that would be important, but currently not in the GVGP domain yet, is the ability to 'remember' games that the agent has played before. Learning the game while playing is a great thing, but this is not something humans do each time from scratch. Finally, it is currently possible to do simulations of the games, as is essential for the MCTS algorithm. But providing that ability makes the benchmark for real AI more questionable. But as AIs get better, those simulations could theoretically be done internally, but this isn't done as of yet.

3.2 The GVGAI Competition

The GVGAI competition is available online and a Java based framework is provided so that anyone can submit their bot. The framework is based on the works of Tom Schaul, who created a Video Game Description Language (VGDL) [35] [36]. VGDL is a language that can describe a simple game easily, usually in less than 50 lines.

3.2.1 Games

There are three sets of 10 games available for the competition: the training set, the validation set and the test set. The first one is the only one where all the games are openly known. This set is used to train and prepare the bots. The second and third are not publicly available. The validation set can be used as a hidden set of games to compare performances between competitors. The test set is used for final evaluation of the competition. Each set has 10 games and there are 5 different levels available for each of game. The training set of the GVGAI 2014 competition is the set of games that is used in the rest of the paper. This is done because Perez et al. [4] also used this set, so results can be easily compared. All the games from the training set are described in Table 3.1 and example screen-shots are visible in Appendix A.

Most games have a non-deterministic element in the behaviour of their entities (NPCs), producing different play-outs each time the same game and level is played. The maximum amount of time steps allowed for each game is 2000 and going over this amount will usually result in a loss (survival games being the exception). This has been done so all games will end, even with a degenerate bot. The games in the training set vary on different aspects. Not only are there different actions available for each game, but also the winning conditions, the scoring mechanisms, and the amount of objects and non-player characters (NPCs) vary. In some games it might be desirable to collide with NPCs (Butterflies, Chase), in others it means game over (Portals, Zelda, and Chase again). Sometimes it is required to quickly finish the game (Missile Command, Butterflies), other times the purpose is to survive as long as possible (Survive Zombies). Clearly, the games require very different strategies to win them. This means implementing game-specific heuristics will be very hard, and probably impossible for the (unknown) games in the other sets. But of course, that is the whole point of General Video Game Playing.

3.2.2 The Framework

As stated, the framework is Java based and can be downloaded [33], so everything can be set up locally. There are some sample controllers provided with the framework, such

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 goats 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.
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 3.1: The 10 games in the training set of the 2014 GVGAI competition [33]. These are the games used for the experiments in this thesis. Descriptions taken from [4]. Screenshots can be seen in Appendix A

as a vanilla MCTS controller, a one step look ahead controller, and a genetic algorithm controller. These are quite simple and only implement the basics of each algorithm, but do give a good idea of the different behaviours of the agents.

Controllers must implement two methods: a constructor for initializing the agent and an `act` method to determine the action for the agent each time step. The constructor is called once at the start and must be completed within 1 second before disqualification. The `act` method only has a time budget of 40 milliseconds. Responding 10 ms late will result in no action being executed, but taking longer than 50 ms results in disqualification. Both methods receive a timer for keeping track of their response times, and a `StateObservation` object that represents the current state of the game.

The `StateObservation` object provides information about the game state, such as the current time step, the available actions, game score, and whether the game is over (win, loss, or ongoing). Two additional pieces of information are also provided to the agent:

- A history of avatar events, sorted by time step, that have happened in the game so far. An avatar event is a collision with the avatar, which is the sprite that represents the agent, and another sprite. Also a collision with a sprite spawned by the avatar (sword, bullet, etc) is included in this list.
- A list of observations and distances to the different sprite types. These are grouped by the sprite category, which is the apparent behaviour of the sprite, such as NPC, collectable, portal, etc.

One essential functionality the `StateObservation` object provides is the `advance` method: the state can be advanced given an action. This means cloning the current state provides a way to simulate the game further, as is of course needed for the MCTS algorithm. However, some games are stochastic, or more precisely, the games with NPCs are stochastic. So it is up to the agent how much to trust the simulated next game states.

Chapter 4

Implementation Analysis

In this chapter the implementations of the vanilla MCTS and the KB Fast-Evo algorithms will be analyzed. It is important to do this, so a foundation is laid down upon which we can try to build a better performing algorithm for GVGP. Comparisons will be made with the results from the paper by Perez et al. [4], which was the paper where the KB Fast-Evo was originally proposed. All the experiments in this chapter were run in the same fashion as in [4]. The framework from the GVGAI Competition [33] was used (see Section 3.2), with the training set from the 2014 GVGAI Competition, as listed in Table 3.1. Every game is played 25 times; each of the 5 levels is played 5 times. So for each implementation, a total of 250 games is played.

A few things should be noted when looking at the result tables. Firstly, comparing score results between the games doesn't give much information, because every game has a different scoring system. This is still included in the tables however, because comparing the scores of the same game between different implementations *does* give meaningful information about performance. Secondly, it turned out that number of roll-outs in a game is relatively low at the start of the game, and goes up as the game goes on longer. This means the average number of roll-outs as seen in the table might be misleading. Games such as Sokoban or Portals take very long when the agent performs badly, but are over very quickly if the agent performs well. So the reader must keep this in mind, but it will be noted again if the average number of roll-outs is biased a lot between implementations.

4.1 Vanilla MCTS

Firstly the Vanilla MCTS implementation was tested. This controller was provided by the GVGAI Competition [33] itself. It uses UCT with $C = \sqrt{2}$ in the UCB1 formula (see

Game	**Win %**	**Avg. score**	**Min. score**	**Max. score**	**Avg. roll-outs**
Aliens	100.0%	64.00 ± 13.35	40	82	139.65
Boulderdash	44.0%	18.12 ± 6.86	3	33	36.28
Butterflies	92.0%	30.24 ± 16.57	10	66	153.02
Chase	48.0%	6.08 ± 3.55	0	15	132.57
Frogs	24%	-0.64 ± 1.29	-2	1	62.45
Missile Command	60.0%	3.64 ± 5.31	-3	16	147.71
Portals	4.0%	0.04 ± 0.20	0	1	142.23
Sokoban	12.0%	0.96 ± 0.89	0	3	204.37
Survive Zombies	56.0%	16.48 ± 12.47	0	40	61.84
Zelda	24.0%	1.36 ± 1.96	-1	6	150.92
Average	46.4%	14.03 ± 2.24			123.10

TABLE 4.1: Results of the vanilla MCTS implementation.

Section 2.4), a maximum play-out depth of 10, node score values are normalized, final action selection is based on the most visits, and a state with a win or loss get a large positive or large negative score respectively. The full algorithm is described in Section 2.2. This same implementation was tested by Perez et al. [4]. Results from the vanilla MCTS test can be seen in Table 4.1.

When looking at the results, something stands out immediately when compared to the results shown in the paper by Perez et al. [4]. The vanilla MCTS algorithm performs *much* better than the results in that paper. Especially Aliens, Boulderdash, Chase and Missile Command have a significantly higher win percentage. Aliens goes from 8% in the results by Perez et al. to 100% here. What causes this difference is uncertain.

4.2 KB Fast-Evo

Next the original KB Fast-Evo as described in [4] was implemented and tested. Results are visible in Table 4.2. These results are quite similar to the results by Perez et al., as opposed to the above results. Only the win percentages of Sokoban and Missile Command differ relatively much, but both results show an improvement over vanilla MCTS. What causes the difference between the results from Missile Command is hard to say. The games are usually over very fast, so the amount of roll-outs per action should have a bigger impact on this game than on other games. The different systems on which the algorithms were tested thus might cause the difference. As for the game Sokoban, the problem space lies largely outside of what the current implementation is able to solve. As Perez et al. also stated, for this game the direction of collisions should

Game	Win %	Avg. score	Min. score	Max. score	Avg. roll-outs
Aliens	100.0%	66.92 ± 12.51	44	81	98.86
Boulderdash	24.0%	20.24 ± 11.00	7	47	26.58
Butterflies	92.0%	26.96 ± 13.44	10	52	106.12
Chase	88.0%	8.80 ± 2.40	6	15	102.58
Frogs	28.0%	-1.00 ± 1.38	-2	1	54.95
Missile Command	92.0%	7.00 ± 5.48	-3	16	110.51
Portals	8.0%	0.08 ± 0.28	0	1	118.42
Sokoban	32.0%	1.24 ± 1.13	0	3	139.65
Survive Zombies	44.0%	19.40 ± 14.54	2	47	45.22
Zelda	32.0%	1.52 ± 1.69	-1	5	106.79
Average	54.0%	15.12 ± 6.38			90.97

Table 4.2: Results of the KB Fast-Evo implementation.

be taken into account, which doesn't happen at the moment. So it is hard to tie a meaningful conclusion to higher or lower win percentages for this game.

4.2.1 Comparing With Vanilla MCTS

KB Fast-Evo performs a lot better overall than vanilla MCTS. In particular Chase and Missile Command perform particularly well. The cause is the agent being drawn *much* more towards 'good' sprites, where the vanilla MCTS agent would often walk around aimlessly. The average number of roll-outs per action are quite a bit lower for the KB Fast-Evo implementation in comparison with the vanilla MCTS implementation. This means that KB Fast-Evo slows the algorithm down, but is to be expected, as a lot of extra calculations happen. One question that could be asked is: How much does the lower amount of roll-outs influence performance? In Section 6.1 this question answered in-depth. But for now we can conclude that improving the algorithm with more calculations, at the cost of a lower amount of roll-outs per action, still increases overall performance. This is visible when comparing above tables.

There are two games however on which the vanilla implementation performs better. Those are Boulderdash and Survive Zombies. The first could be explained with the fact that the agent is drawn to the score-boosting diamonds, but often gets stuck behind some obstacles while keeping to try and reach the nearby diamond. The vanilla MCTS agent walks around more randomly, so is less likely to get stuck in that way. Survive zombies has a similar problem, but also the lower amount of roll-outs influences this game. Finding the very best path past the zombies and towards the honey is crucial, but is more likely to fail with a fewer roll-outs.

4.2.2 Analysis Of Strengths And Flaws

In this section the KB Fast-Evo algorithm will be analyzed more thoroughly. After implementation, some things were noticed that did not seem to work very well in practise, and other things worked as expected. Keep in mind that this analysis is purely based on observations seen from the implementation of the KB Fast-Evo algorithm from Perez et al. [4]. The analysis is split in two parts, based on where the KB Fast-Evo algorithm differs the most from the original MCTS algorithm. Some thoughts for improvements will be given as well.

4.2.2.1 The Scoring Function

One of the biggest changes in the KB Fast-Evo algorithm is the use of a knowledge base in the scoring function. The scoring function is used to update node values in the tree, which influences the paths walked through during the tree policy, and ultimately the chosen action. Note that this scoring function is also used for evaluation of the weight matrices that bias action selection in roll-outs. This part will be discussed in the next section.

In GVGP the end state of the game will usually not be reached during a roll-out. That's why a scoring function has to be used so that each roll-out will still have a score (rather than score of 1 for a game win and a score of -1 for a game loss). In the vanilla MCTS implementation, this scoring function is simply the scoring function of the game itself. So for instance in Aliens, you get a +2 score whenever you shoot an alien.

As mentioned before, there is a problem with using this scoring function. When a score change is beyond the horizon reachable by the roll-outs in the MCTS algorithm (10 steps in this case), the algorithm is actually just selecting actions at random. This is clearly visible in the game Butterflies: when the agent is near a butterfly it goes straight for it, because the butterfly gives a score boost. This happens because a lot of the roll-outs have provided a positive score change and thus the tree policy guides the agent to the butterfly. However when the agent is not near any of the butterflies, it will just randomly walk around until a butterfly is within reach again. In practise 'within reach' means around 3–4 steps away because the roll-outs are random, so they have to randomly reach the butterfly from their starting point.

The solution proposed by Perez et al. [4] introduces a knowledge base and a different scoring function based on the knowledge base. The basic idea was already explained in Section 2.5.12, but the scoring function will be given in full here:

$$Reward = \begin{cases} \Delta R & : \Delta R \neq 0 \\ \alpha \times \Delta Z + \beta \times \Delta D & : \text{Otherwise} \end{cases} \tag{4.1}$$

where ΔR is the score change, ΔZ the knowledge change, and ΔD the distance change. The latter two values are calculated as follows:

$$\Delta Z = \sum_{i=1}^{N} \Delta(K_i) \tag{4.2}$$

$$\Delta(K_i) = \begin{cases} Z_{iF} & : Z_{i0} = 0 \\ \frac{Z_{iF}}{Z_{i0}} - 1 & : \text{Otherwise} \end{cases} \tag{4.3}$$

$$\Delta D = \sum_{i=1}^{N} \Delta(D_i) \tag{4.4}$$

$$\Delta(D_i) = \begin{cases} 1 - \frac{D_{iF}}{D_{i0}} & : Z_{i0} = 0 \ OR \ D_{i0} > 0 \ and \ \overline{x_i} > 0 \\ 0 & : \text{Otherwise} \end{cases} \tag{4.5}$$

Where Z_{i0} is the number of occurrences of event i at the beginning of the play-out and Z_{iF} the number of occurrences at the end of the play-out. D_{i0} is the euclidean distance to the closest sprite of type i at the beginning of the play-out, D_{iF} is the euclidean distance at the end of the play-out. $\overline{x_i}$ is the average score change of event i.

So for the final reward score, if there is a difference in score before and after the roll-out, the score value is used, just like in the vanilla algorithm. But if there is no score change, the values ΔZ (knowledge change) and ΔD (distance change) are used. The knowledge base stores information about events (collisions): How often they have occurred, what the score change was, and what sprites have collided. This information is used to calculate the two values above. Essentially, the knowledge change will be higher when a roll-out has produced more events, especially when those events have rarely occurred. The distance change will be higher when the agent has reduced its distance from unknown sprites or sprites that have provided a score boost in the past. In Perez et al. [4], the values $\alpha = 0.66$ and $\beta = 0.33$ are used, which were determined empirically. The sames values are used in this thesis.

This *does* solve the problem mentioned above, namely that the agent no longer moves around randomly when there is no score change within his reach. It's clearly visible that the agent moves towards interesting locations. But there are some problems that occur when using this.

First of all, in most cases, the ΔZ knowledge change part does not add a lot of information. The knowledge change only applies to events that do not result in a score change, because otherwise the normal ΔR score change is used. In practise most events result in a score change and other events aren't very interesting. Also, the value quickly converges very close to zero. It is not per se bad to use this, but in this form, the agent is only stimulated to pick an action that leads to a lot of different events (that are without score change, and rarely seen) *within its reach.* It should be much more beneficial to direct the agent towards events that haven't occurred (often) yet, especially when the game lasts longer.

The distance change value works well in some cases; the agent no longer walks around randomly when there is no score-boosting sprite nearby. The agent now moves towards other sprites, even if he is far away. However, there are still some issues with this. The first problem is not a very big one, but still causes problems sometimes. Namely when there are two different sprites which give a score boost (or are undiscovered) on either side of the avatar, the distance change value is essentially non-effective at that point. This could be solved by prioritizing one sprite type over the other.

The other problem results in more serious issues, which can clearly be seen in the game Butterflies, level 5 (see Figure 4.1) There are some sprites that can never be reached because they are behind a wall. When the agent is near that wall and all the butterflies are far away, the agent will stick to the wall because the distance change value will direct it to the sprites that can't be reached. The issue also arises when the agent is stuck somewhere behind a wall. This is all because distance is measured in euclidean distance and obstacles aren't taken into account. One way to solve this (as also proposed by Perez et al. [4]) is using a simple path finding algorithm to work around obstacles. The question remains if it will actually improve the algorithm, because it will make it slower, which will of course result is less roll-outs. Also in some games (Boulderdash), you can remove obstacles, so the path finding algorithm must be able to work with that. A less effective, but more simple solution might be to take actions out of consideration that are not possible because of obstacles. This will increase the chance that the agent will go out of the local optimum, but the problem essentially will be still there.

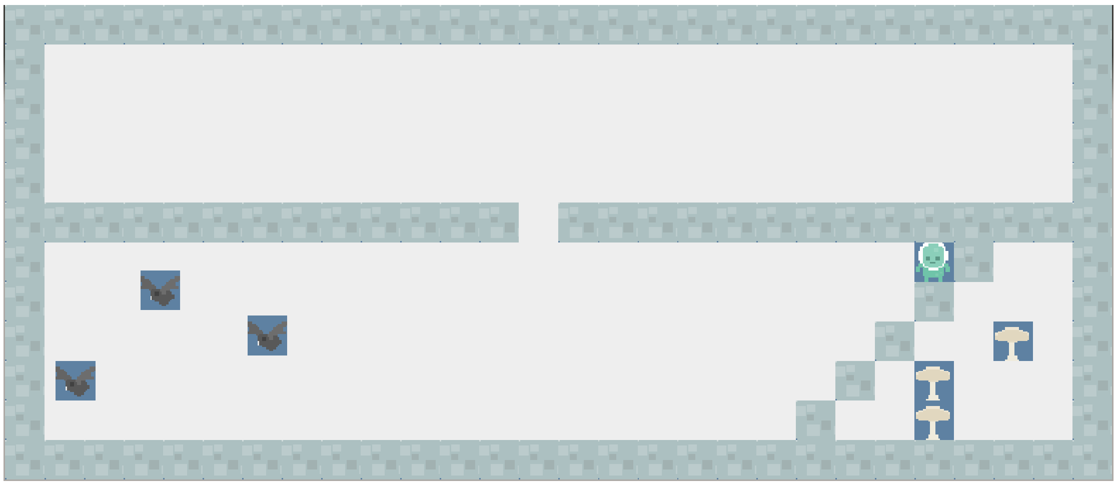

FIGURE 4.1: In Butterflies level 5, the avatar sticks to the wall because it is trying to reach sprites that cannot be reached.

4.2.2.2 Roll-outs

One of the problems with the vanilla MCTS algorithm is that roll-outs are random. This means that not every roll-out produces information that is useful. This problem goes especially for GVGP; because of the small amount of time per move there are fewer roll-outs. The KB Fast-Evo algorithm introduces a function that uses features and a weight matrix to bias action selection in the roll-outs, so that they will be guided in a more meaningful direction. This is done by first giving each available action a value a_i

$$a_i = \sum_{j=1}^{N} w_{ij} \times f_j \tag{4.6}$$

w_{ij} is the weight in the weight matrix for action i and feature j. f_j is the feature value. Then this value is used to select each action with a chance $P(a_i)$, using Gibbs sampling:

$$P(a_i) = \frac{e^{-a_i}}{\sum_{j=1}^{A} e^{-a_j}} \tag{4.7}$$

Features are sprites such as NPCs, portals, resources, etc. The feature value is the euclidean distance to the closest sprite of each type. The weight matrix contains a weight for each feature in combination with each possible action. Initially it has weights of 1, but it's evolved during the game using a simple (1 + 1) evolutionary strategy and the scoring function explained in the previous section as fitness function.

First the feature value will be analyzed. Sadly, Perez et al. [4] do not mention the reasoning behind taking the euclidean distance to each sprite type as a feature value.

However, it is clear why distance is important, because interesting things happen usually when events are produced. Guiding roll-outs towards events will result in a higher chance for useful information at the end of the roll-out. But is the euclidean distance a good value? That seemed illogical at first, because the sprite types that are the furthest away receive the highest value. Tests were ran with a new feature value, namely $\frac{1}{f_j}$, so that the closest sprites get the highest values. This resulted in the agents wanting to stay near a close sprite way too much. The original feature value performed better by far. The exact reason behind taking this feature value (not taking the weight matrix into account) could be researched further, but is beyond the scope of this thesis.

Secondly, the weight matrix and its evolution will be discussed. At first, this seemed like a great idea to bias the roll-outs in a meaningful way, but after implementation a few issues arose. As said, the weight matrix contains a weight for each available action (i.e. right, left, up, down and use), in combination with each feature. But there is a serious problem with this. To illustrate this, let's take the game Zelda for instance. In that game, it is important to do the action 'use', on a monster sprite in order to kill it and receive points. Let's say the matrix has evolved to learn this and the weight for action use with the monster sprite is relatively high. But when the agent is next to a monster, its feature value (distance) is low. Now looking at the Formula 4.6, all the weights for action 'use' are multiplied with every feature value and then added up to get the action value. But as the monster is close and the feature value is low, the high value of the weight will be lost in the total sum with the features that are far away. So the current feature value in combination with the weight matrix might not be the best way to approach this.

There are more issues. Move actions are very relative to the current state of the game. If the agent starts at the left side of the map, it would quickly learn that going to the right is a good thing to do. This is of course because going to the right will likely produce more events, rather than going to the left (where the edge of the map is). But after a few game steps, it is entirely possible that the agent has moved all the way to the right side of the map. However, the weights for the actions are still there, meaning that it has a high preference to go to the right rather than to the left. Then it has to learn again that going to the left is better. In conclusion: with the move actions, the weight matrix is always a few steps behind the actual state of the game. This does not go for the 'use' action, as for instance killing a monster with your 'use' action will always be a good thing to do throughout the game. So a solution to the problem could be taking all the separate move actions out of the matrix and putting in one 'move' action. This weight then represents how desirable it is to go towards a certain feature. This takes out the relative position issue mentioned.

There is one final issue that arose when the KB Fast-Evo algorithm was implemented: Once the scoring function reaches its highest value, the matrix is no longer evolved. This is a problem because the value of the scoring function is completely dependant on the state of the game. Let's say in Butterflies, the agent manages to capture 3 butterflies in one roll-out, early in the game. This leads to a score of 6, which is never improved upon, because there are less and less butterflies. So now the matrix that was used in that roll-out is always picked, evolved, and its child will always have a lower fitness and is thrown away again. The agent can no longer learn. In the implementation it becomes clear that the matrix very quickly converges to its final form. So there should be some solution to this problem that will keep high scoring matrices relevant, but at the same time makes sure the agent can still learn. The next Chapter 5 will try and solve most of the issues mentioned in the above analysis in order the improve the KB-Fast Evo algorithm as much as possible.

Chapter 5

Improving KB-Fast Evo

As can be read in Chapter 4, the KB Fast-Evo algorithm still has room for improvement. In this chapter some of the suggestions from Chapter 4 will be implemented, tested and analyzed. The idea is to try and make the original KB Fast-Evo algorithm from Perez et al. [4] perform better for the GVGP platform. Each section will contain one specific idea that was implemented, and then analyzed by using its test results.

The different implementations are tested in the same way as in Chapter 4, i.e. every one of the 5 levels of a game is played 5 times, for a total of 25 runs per game.

5.1 Adding A Path Finding Algorithm

The first thing that was implemented is a path-finding algorithm. This was also suggested by Perez et al. in their paper [4]. In Sections 4.2.2.1 and 4.2.2.2 can be read that in the original KB Fast-Evo algorithm, the euclidean distance from the avatar to different sprite types (features) was used two times: once for the distance change value and once for the feature value. The first is used in the scoring function and the second is used to guide the roll-outs. Problems with the euclidean distance arise when there are obstacles preventing the avatar to reach the feature directly. And even worse, sometimes features can't be reached at all. A simple A* path-finding algorithm was implemented to solve these problems, where euclidean distance was replaced by the actual distance taking obstacles into account. The obstacles, or non-traversables are just the walls. Results of this implementation can be seen in Table 5.1

The results show a slight improvement to the overall win-percentage. Some games benefit clearly from the path finding, such as Butterflies, Chase and Zelda. This was to be expected as the levels in those games contain a lot of walls which the agent has

Game	Win %	Avg. score	Min. score	Max. score	Avg. roll-outs
Aliens	100.0%	63.76 ± 12.41	40	77	37.37
Boulderdash	32.0%	14.56 ± 8.75	3	29	16.92
Butterflies	100.0%	30.16 ± 16.16	14	70	58.74
Chase	96.0%	9.12 ± 2.76	6	15	60.83
Frogs	16.0%	-1.36 ± 1.18	-2	1	32.01
Missile Command	88.0%	6.52 ± 5.01	-1	16	71.90
Portals	12.0%	0.12 ± 0.33	0	1	34.26
Sokoban	20.0%	1.04 ± 0.93	0	3	109.87
Survive Zombies	40.0%	20.88 ± 16.32	-1	49	31.58
Zelda	48.0%	1.96 ± 2.03	-1	6	57.52
Average	55.2%	14.68 ± 6.59			51.11

Table 5.1: Results of implementing a simple A* path-finding algorithm. This replaces the euclidean distance used in the original KB Fast-Evo algorithm.

to move through. Adding the path finding algorithm helps the agent not getting stuck anymore. Games such as Portals and Sokoban also have a lot of walls, but the algorithm currently cannot handle those games very well, so adding the path finding does not help much.

One other thing that stands out, also as expected, is the lower amount of roll-outs per action. On average for all the games, the amount of roll-outs is 43.82% lower than without the path finding. This hurts the performance of other games, such as Frogs, where the agent is much more likely now to make a bad decision and die while crossing the road. More on the importance of the (amount of) roll-outs in Section 6.1

Because this was the most basic implementation of A* into the KB Fast-Evo algorithm, there are still ways to improve the algorithm to make it faster. This is mostly because it is used on all the features each time the avatar makes a move in the actual game or in a roll-out. This quickly adds up to a lot of path finding per single move.

One way to improve is simply to not use the path finding when it is not necessary, so when there are no obstacles in the level (Missile command), or when the agent has only two move actions (Aliens). Another improvement could be identifying sprites that cannot be reached and not calculating them again. The path finding algorithm will be the slowest for those sprites, because it will search across the whole map before giving up.

In order to reduce the amount of times the path finding algorithm is called, we could only calculate it at the start and the end of a roll-out, which means we calculate it ~82%

(9 out of 11, if roll-out depth is 10) less. This means that the feature value needs to be estimated from the distance calculated the first time. By using the euclidean distance when the the actual distance and the euclidean distance are about the same this is partly achieved. When the actual distance and the euclidean distance are not the same, we can decrease distance when the agent follows the path, and increase it otherwise. The analysis from Chapter 4 also suggested removing move actions from consideration that have no effect (i.e. going right when there is a wall on the right side).

After implementing the above, it turned out that the path finding in the roll-outs did not slow the algorithm down too much anymore. The more inaccurate improvement of calculating the path only at the start and end of the roll-out will not have be implemented. So even the roll-outs now have an accurate distance value, where the path is calculated each time again. Removing useless move actions from consideration *did* turn out to take long in the *roll-outs*, because a check had to be performed whether the agent had moved, which is a slow operation. So tests were performed with removing useless move actions from consideration *only* when expanding the tree. Results from that have shown to not only not improve the algorithm speed much (only ~3% faster), but in some games (Boulderdash, Chase, Frogs) it hurt performance. This was because the agent was more likely do collide with an NPC and die when cornered. So improving the path finding algorithm is now only done with the following:

- Not using path finding when it's not needed (i.e. there are no obstacles on the map).
- Not calculating feature distance for features that cannot be reached.

The results for this implementation can be seen in Table 5.2.

The overall performance is slightly better than without the optimization, both in average score and the win percentage. The average amount of roll-outs however increased by 41.09%, which is a significant increase. Most games perform about the same as without the optimization to the path finding. There is one game that has a slightly lower win percentage, which is Butterflies. The average score for the game is slightly higher however, so overall performance is the about same. Two games perform clearly better with the optimization, and that is due to the increased amount of roll-outs. Those games are Frogs and Missile Command. As we saw before, Missile Command is sensitive for a lower amount of roll-outs because the games are over very fast. Also Frogs is influenced by a lower amount because the agent is more likely to collide with an NPC. Something that could still improve the path finding algorithm is letting the agent learn which sprites are actual obstacles besides the walls. In Boulderdash for instance the agent cannot move through the boulders, but can 'dig' through the dirt (see Figure A.2).

Game	Win %	Avg. score	Min. score	Max. score	Avg. roll-outs
Aliens	100.0%	67 ± 13.25	40	81	99.69
Boulderdash	32.0%	18.56 ± 12.33	3	55	20.46
Butterflies	92.0%	30.80 ± 17.25	14	72	72.29
Chase	92.0%	9.24 ± 3.35	5	15	72.76
Frogs	24.0%	-0.96 ± 1.34	-2	1	37.83
Missile Command	96.0%	7.40 ± 5.04	1	16	107.13
Portals	16.0%	0.16 ± 0.37	0	1	81.35
Sokoban	24.0%	1.36 ± 0.86	0	3	123.67
Survive Zombies	40.0%	18.16 ± 11.90	3	38	36.11
Zelda	44.0%	2.44 ± 1.92	-1	6	69.75
Average	56.0%	15.42 ± 6.76			72.11

Table 5.2: Results of implementing a simple A* path-finding algorithm, with improvements to efficiency.

5.2 Changing The Evolutionary Approach

The next thing that was implemented in order to try and solve a lot of the issues mentioned in the analysis was a different evolutionary algorithm. One of the main things that was changed is the weight matrix. The problem of having all move actions separately in the weight matrix was mentioned in the analysis from Chapter 4, so now all different move actions are combined into one move action. The 'use' action remains the same. The way the matrix is used now will be described here, as some essential parts have changed.

The weight matrix contains weights for each action (so at most 'move' and 'use'), in combination with each feature in the game. The action value for a move is the sum of all weights for that action. In the roll-out the action is still chosen with Gibbs sampling, according to the relative height of the action value, as in the paper. Now there is of course a difficulty in getting the actual value for a certain move action, because in the matrix there is only one move weight. This was solved by getting the distance change per move action for each feature. The following formulas were used.

$$a_i = \sum_{i=1}^{N} \left(w'_{ij} \times \frac{f_j}{0.1t} \right) \tag{5.1}$$

$$w'_{ij} = (1 - d_{ij}) + (d_{ij} \times w_{ij}) \tag{5.2}$$

w'_{ij} is the new weight for action i and feature j, which is calculated with the weight w_{ij} from the matrix and d_{ij}. The latter is the normalized distance change value to feature j when action i is used. So let's say a feature j is directly right of the agent, then d_{ij} is 1 for action 'right' (maximally decreased distance to feature j), 0 for action 'left' (maximally increased distance to feature j), and some value in between for the 'up' and 'down' actions. So basically for w'_{ij}, the more an action directs the agent towards the feature, the more of the actual weight for that feature is used. The weights w_{ij} for the different move actions i for a certain feature j are now of course the same value, because move actions are combined into one in the matrix.

The action value a_i is calculated with this adapted weight and with the feature value f_j, but the other change from the original formula is that the feature value f_j is divided by a factor of the current time step t. The feature value f_j is still the distance to the closest sprite of that feature type, as in the original KB Fast-Evo algorithm. Experiments were done without the feature value, only taking the weight w'_{ij}, in order to solve the issue mentioned in Section 4.2.2.2: the feature value cancels out the weight. This turned out not to work very well. The agent takes a while to learn by evolving the matrix, while the feature value directly gives a good working heuristic to bias the roll-outs. That is why the feature value f_j is now divided by a factor of the current time step t. As the matrix evolves in the right direction, it becomes more important for the action value.

One of the other problems mentioned was that the matrices would reach a point where they no longer evolved, because the maximum fitness value in a game was reached. To solve this, the new implementation no longer uses a $(1 + 1)$ evolutionary strategy, where the population has a maximum size of two. Now, for each roll-out a new matrix is copied and mutated from one parent matrix and used in that roll-out. The fitness value for that matrix is the scoring function, as also in the original KB Fast-Evo algorithm. New matrices are added to the population until the MCTS algorithm is done and has selected an action. When the MCTS algorithm starts up again, it sorts all the matrices according to their fitness and selects a new parent and throws away all other matrices. The parent is selected with a high chance to be the matrix with the highest fitness in the population, and with lower chances to be the next few matrices with lower fitness. Selecting the parent this way will result in different parent matrices throughout the game, as opposed to the same parent for the most part of the game.

The final change in the evolutionary part of the algorithm is the way the matrices are evolved. In the KB Fast-Evo implementation a Gaussian distribution with a mean of 0 and a standard deviation of 0.1 was used to get a value to add to a weight of the matrix. Each weight had a 0.2 chance to be mutated this way. Now, in order to evolve the matrices faster in the right direction, we use knowledge from the game that was already present in the knowledge base. If an event (i.e. collision with a certain feature) has occurred more than 100 times, we have a good indication of the score change this event has produced. If the score change is positive, we change the mean of the Gaussian distribution up and if the score change is negative, the mean goes down. This causes the weights to be stimulated upwards for 'good' features and downwards for 'bad' features.

All the changes to the Evolutionary part of the KB Fast-Evo algorithm are listed below.

- All move actions are combined in the matrix into one. The individual move action weights are calculated with the distance change to the features.
- The feature value used in calculation for the action values is divided by a factor of the current time step.
- The population is grown during roll-outs and selection only happens at the start of a MCTS run.
- The parent matrix is selected with a chance according to its fitness.
- Matrix mutation is biased by the knowledge base.

The results can be seen in Table 5.3.

We can see that the algorithm performs only slightly better than original KB Fast-Evo algorithm. This is an unexpected result, because as we saw in the analysis, the evolutionary part in the KB Fast-Evo algorithm hardly influenced the game outcome. That was because the weight matrix very often stopped evolving in the beginning of

Game	**Win %**	**Avg. score**	**Min. score**	**Max. score**	**Avg. roll-outs**
Aliens	100.0%	67.96 ± 13.14	42	81	97.44
Boulderdash	36.0%	18.00 ± 6.73	6	30	27.67
Butterflies	92.0%	30.56 ± 18.24	0	80	109.17
Chase	88.0%	8.96 ± 2.70	5	15	100.63
Frogs	8.0%	-1.52 ± 1.00	-2	1	99.94
Missile Command	92.0%	6.20 ± 5.50	-2	16	102.50
Portals	12.0%	0.12 ± 0.33	0	1	115.47
Sokoban	24.0%	1.08 ± 0.95	0	3	145.62
Survive Zombies	44.0%	17.60 ± 16.82	-1	57	50.29
Zelda	64.0%	2.52 ± 2.10	-1	6	114.33
Average	56.0%	15.15 ± 6.75			96.30

Table 5.3: Results of implementing the improved evolutionary approach.

the game. As this problem was now solved, with a weight matrix that keeps evolving throughout the game, it *does* influence the action selection. Even though this new evolutionary approach didn't change the game results very much overall, there are still a few differences.

What could be noticed firstly, is that the average number of roll-outs is actually higher than in the original algorithm. Because the new implementation has no new heavy calculations that would slow it down, we could expect that the algorithm would not be slower. But the reason that it seems faster has to do with what was mentioned earlier: the average number of roll-outs goes up as the games last longer. So actually the algorithm is not faster, but rather the agent takes longer to finish (or fail) some games. This is particularly visible in the game Frogs, which will be analyzed now.

There is one game which has a significant lower win-percentage with the new evolutionary implementation when compared to the original KB Fast-Evo. That is Frogs, but the reason seems clear. The agent gets 'scared' of crossing the road because of the evolution and will not try it anymore and stick to the starting area. The agent learns quickly from its simulations now, and in those simulations he dies often to the trucks on the road. The weight will drop for the feature that represents those trucks. While the agent seldom reaches the end point of the game in its simulations, so that weight will evolve more slowly. This is also why the average number of roll-outs is a lot higher in this implementation, because the game is mostly lost after the time-out of 2000 time steps, where in the KB Fast-Evo algorithm the agent sometimes died early while crossing the road.

The game that performs a lot better is Zelda. The reason is less obvious than it was for Frogs, but can be explained anyway. Firstly, the agent is able to avoid dying to the NPCs better, because it will actively move away from them after the matrix had a chance to evolve. Secondly, the agent is better at finding the key after a while, because it gives a score boost and thus is more attractive. So the new evolutionary approach is at least able to make a big difference for one game. But biasing the roll-outs does not influence performance as much as first thought.

5.3 Combining Path Finding And The New Evolutionary Approach

It would of course also be interesting to see what happens when we apply both the improvements to the KB Fast-Evo algorithm and see how that improves the performance. Those results can be seen in 5.4.

Game	Win %	Avg. score	Min. score	Max. score	Avg. roll-outs
Aliens	100.0%	67.60 ± 12.85	43	80	101.52
Boulderdash	24.0%	10.88 ± 10.24	-1	36	20.48
Butterflies	100.0%	28.08 ± 13.50	14	56	75.68
Chase	92.0%	9.20 ± 2.66	6	15	71.71
Frogs	16.0%	-1.04 ± 1.24	-2	1	75.16
Missile Command	92.0%	5.76 ± 5.70	-3	16	102.11
Portals	20.0%	0.2 ± 0.41	0	1	89.76
Sokoban	16.0%	1.04 ± 0.98	0	3	117.03
Survive Zombies	44.0%	20.88 ± 17.38	1	51	35.84
Zelda	60.0%	2.24 ± 2.24	-1	6	67.58
Average	56.4%	14.48 ± 6.71			75.69

TABLE 5.4: Results of applying the path finding algorithm *and* the new evolutionary approach to the KB Fast-Evo algorithm.

The combined improvements do not enhance the algorithm as much as expected, but there are some interesting things to be noticed. Firstly, because of the path finding, the algorithm is slower again. More specifically, the algorithm is ~20% slower than the KB Fast-Evo with only the changed evolutionary approach. Especially Zelda and Butterflies are a lot slower, but those games both benefit from the path finding because they have obstacles. In fact, Butterflies is the one game that actually performs very well with this implementation, reaching a winning percentage of 100%.

The other games that performed well with the path finding still perform well here. Frogs, the game that performed worse with the new evolutionary approach, gets some better results again. The same goes for the games that performed well with the new evolutionary approach. They still perform well here. As before there are still games that are hard for the algorithm, like Portals and Sokoban.

All in all we can conclude that the combination of the improvements shows the benefits from the separate improvements, but the overall increase in performance is not much higher. Some better results were maybe expected when the KB Fast-Evo algorithm was analyzed in Chapter 4, but still there is an increase in performance with the above implemented as compared to the original KB Fast-Evo algorithm. In Section 6.3 some additional (more general) improvements will be implemented, to see if the algorithm can be enhanced some more.

Chapter 6

More Improvements And Experiments

This chapter will be the final chapter where experiments will be performed with the KB Fast-Evo algorithm. The chapter firstly aims to show the influence of changing some of the default settings for the algorithm that were used previously in the thesis. Secondly, there will be a final attempt to maximize the performance of the KB Fast-Evo algorithm. Some smaller improvements based on the analysis from Chapter 4 and some improvements based on observations will be implemented and tested.

The first section will test different roll-out settings and will confirm claims made earlier about the influence of (less) roll-outs. The second section will show what happens when the constant C parameter from the UCT algorithm is changed, which so far had been assumed to be optimal. Finally, the last section will show the implementation and results from the final improved KB Fast-Evo algorithm.

6.1 Importance Of The Roll-outs

In the MCTS algorithm, a lot is dependant on the roll-outs. The roll-outs provide data on which states are desirable and which actions lead to those states. The more roll-outs we have, the more certain we can be about the validity of the data. In fact, as stated in Section 2.3, an infinite number of roll-outs would lead to the correct full-depth mini-max tree. This applies only to roll-outs that play out the full game, and not as here, with a limited depth (of 10). The full-depth mini-max tree is the optimal game tree, so can we conclude that more and deeper roll-outs lead to better results? This section will try to answer that for this algorithm.

	Action time 40ms Roll-out depth 10		**Action time 80ms Roll-out depth 10**	
Game	**Win %**	**Avg. roll-outs**	**Win %**	**Avg. roll-outs**
Aliens	100.0%	98.86	100.0%	237.51
Boulderdash	24.0%	26.58	44.0%	60.65
Butterflies	92.0%	106.12	100.0%	248.78
Chase	88.0%	102.58	80.0%	221.40
Frogs	28.0%	54.95	28.0%	115.00
Missile Command	92.0%	110.51	92.0%	266.16
Portals	8.0%	118.42	4.0%	253.97
Sokoban	32.0%	139.65	24.0%	328.84
Survive Zombies	44.0%	45.22	60.0%	107.51
Zelda	32.0%	106.79	32.0%	252.45
Average	54.0%	90.97	56.4%	209.22
	Action time 40ms Roll-out depth 20		**Action time 80ms Roll-out depth 20**	
Aliens	100.0%	53.26	100.0%	116.20
Boulderdash	32.0%	14.97	40.0%	33.40
Butterflies	92.0%	58.15	96.0%	133.48
Chase	80.0%	54.86	80.0%	122.55
Frogs	4.0%	64.81	12.0%	149.56
Missile Command	84.0%	62.91	84.0%	164.73
Portals	16.0%	85.72	16.0%	188.94
Sokoban	36.0%	88.78	24.0%	203.60
Survive Zombies	44.0%	27.37	52.0%	58.81
Zelda	28.0%	66.53	40.0%	142.61
Average	51.2%	57.74	54.4%	124.39

TABLE 6.1: Results of running the KB Fast-Evo algorithm with different action time settings and roll-out depths. Top left results in the table are with the default settings as used in above chapters.

In the chapters above it was often visible that implementations of the algorithm with less roll-outs would lead to worse results for certain games. Here, the amount of roll-outs per action selection will be increased to see what happens then. This is done by increasing the time the algorithm for action selection. The value used in the above chapters was 40ms, as is required for the GVGAI Competition, but will be altered here for the tests. Experiments will also be ran with an increased roll-out depth, to show what influence that has over the performance of the algorithm. The standard KB Fast-Evo from [4] will be used so good comparisons can be made. All results are visible in Table 6.1

Firstly the KB Fast-Evo algorithm with an increased action time of 80ms is compared with the normal 40ms action time. These can be seen on the top-right of Table 6.1.

What immediately stands out is the large increase in average roll-outs for each game. This is of course the purpose of this experiment and is caused by having twice as long between each move to do the roll-outs. The second thing that is noticeable is that there is not a huge increase in performance of the algorithm. Three games *do* score better, which are Butterflies, Boulderdash and Survive Zombies. This has to do with the fact that the agent has to either find or avoid NPCs. With an increased number of roll-outs, the movements of the NPCs are better to predict and thus the agent survives longer and/or scores better. But overall there is not a much better performance, even though the algorithm runs twice as long. We can draw a good conclusion from this: 40ms is enough for this algorithm to get a good approximation of the action values based on the used scoring function. To confirm this conclusion the other way around, the same experiment was ran except with the action time decreased to 20ms. The average win-rate drops almost 10% and nearly every game performs worse than with an action time of 40ms. So 40ms seems to be correctly chosen for a roll-out depth of 10.

Side note: The same experiment was ran with the vanilla MCTS algorithm instead of the KB Fast-Evo algorithm. These results showed a bigger increase in performance, but as the roll-outs are not guided in the vanilla algorithm, it is logical that there need to be more roll-outs for accurate estimation of the action values. However, the 80ms vanilla algorithm still performed worse than the KB Fast-Evo algorithm with an action time of 40ms.

So increasing the amount of roll-outs does not increase the performance of the algorithm very much. But when the reachable search space is broadened by increasing the roll-out depth, without decreasing the amount of roll-outs, it seems that there should be an increase in performance. This was also tested and can be seen in the bottom half of Table 6.1. This time a roll-out depth of 20 was taken, instead of the default roll-out depth of 10. On the bottom-left, the action time is the default 40ms, and on the bottom-right the action time is 80ms.

With the default action time and the increased roll-out depth, the average amount of roll-outs is significantly lower. This also reflects in the performance of the algorithm, which is lower than with the default settings. This was expected, because as can be read above, decreasing the amount of roll-outs influences the performance in a bad way. But now when looking at the bottom-right of Table 6.1, the average amount of roll-outs isn't lower than with the default settings. In fact, it is higher. Still the average amount of wins is about equal to the normal KB Fast-Evo algorithm. This was quite unexpected, but after some further investigation, can be explained. There are a few games that perform better, which are Boulderdash, Survive Zombies and Zelda. And there are a few games that perform worse, which are Chase, Frogs, Missile Command and Sokoban (see Appendix A for reference). The first set of games that did perform better all have something in common. Namely it is important to find a path to a goal that might be far away. The deeper roll-outs allow this to happen easier, because the agent has a bigger search space and can 'see' the goal from a greater distance. That is why these games perform better.

The games that perform worse also have something in common (Frogs will be mentioned separately). In these games, it is important to quickly go for the nearby score boost, rather than having to find it further away. In Chase, once the goats had the time to run, the agent has a more difficult time finding them and the goats have a chance to kill the agent after a while. In Missile Command it is clear that the missiles need to be reached as soon as possible, before the cities are destroyed. And in Sokoban, which is a hard game for this algorithm, most games are solved when the agent pushes the boxes straight in the holes. The harder levels where more complex movements are required are rarely won. With deeper roll-outs, the agent is less likely to go for nearby score boosts, because it can see more score boosts further away, which draws the agent away from the nearby ones. That's why these games perform less well. Frogs is different in that there are not much score boosts, but the reason it performs worse is the same. Nearly every time the agent dies in the first few game steps. As in the other games, the deeper roll-outs blur the score change for nearby events, so the agent is a lot more likely to die to the cars when he is near to them.

These results were mostly unexpected but interesting nonetheless. One important conclusion that can be drawn from this is that the default settings of 40ms per action (mandatory setting for the competition) and a roll-out depth of 10 are good settings. Also seeing that 40ms currently works well, there is no need to make the algorithm faster than it already is. However, when additional calculations are required in order to make the algorithm more effective, one should try not to lower the current amount of roll-outs too much, as that *does* influence performance negatively.

6.2 Influence Of The C Parameter

Another factor in the performance of the MCTS algorithm, or more particularly, the UCT algorithm, is the tuning of the constant C parameter in the UCT formula. A detailed explanation of this formula and the C parameter were already given in Section 2.4. The C parameter determines the amount of exploration in the *tree policy*. The value used throughout this thesis was $\sqrt{2}$, as also used in the paper by Perez et al. [4]. But different papers have described different parameter values for their algorithms. For instance in high performing algorithms for Go [37] and Hex [38], the C value was 0. These

	$C = 0$		$C = \frac{1}{\sqrt{2}}$	
Game	**Win %**	**Avg. roll-outs**	**Win %**	**Avg. roll-outs**
Aliens	100.0%	104.32	100.0%	100.75
Boulderdash	32.0%	27.13	32.0%	26.76
Butterflies	96.0%	158.53	100.0%	111.98
Chase	96.0%	129.33	88.0%	99.54
Frogs	16.0%	63.66	24.0%	55.71
Missile Command	100.0%	192.07	88.0%	112.11
Portals	12.0%	187.86	4.0%	114.73
Sokoban	16.0%	147.63	20.0%	146.19
Survive Zombies	40.0%	55.86	40.0%	49.49
Zelda	40.0%	125.74	44.0%	113.58
Average	54.80%	119.21	54.0%	93.08
	$C = 1$		$C = 3$	
Aliens	100.0%	102.46	100.0%	100.25
Boulderdash	32.0%	27.49	36.0%	27.91
Butterflies	100.0%	109.24	92.0%	108.74
Chase	84.0%	102.67	88.0%	102.13
Frogs	28.0%	54.03	20.0%	57.84
Missile Command	88.0%	116.24	80.0%	122.92
Portals	20.0%	119.53	12.0%	118.92
Sokoban	16.0%	141.03	12.0%	140.91
Survive Zombies	32.0%	52.07	40.0%	51.67
Zelda	40.0%	113.97	32.0%	115.89
Average	54.0%	93.87	51.20%	94.72

TABLE 6.2: Results of running the KB Fast-Evo algorithm with different C parameter values in the UCT formula. The results with the default value $C = \sqrt{2}$ can be viewed in Table 6.1 or Table 4.2

algorithms completely relied on their history heuristics (AMAF/RAVE) for exploration. Another example is the General Game Playing algorithm CadiaPlayer [19], which used a C value of 0.4.

In this section we will test out 4 different values (besides the default of $\sqrt{2}$) for the C parameter to see which performs the best. These values are 0, $\frac{1}{\sqrt{2}}$ (≈ 0.71), 1, and 3. The value $C = \frac{1}{\sqrt{2}}$ is the value that Kocsis and Szepesvári have shown to satisfy the Hoeffding inequality [5]. Again, the normal KB Fast-Evo algorithm is used. The results can be seen in Table 6.2.

Some interesting facts can be seen when looking at Table 6.2. First of all, the algorithm is faster when $C = 0$: there is a ~11% increase in roll-outs. Secondly, all C values tested result in about the same average win percentage, except the highest $C = 3$. And finally, the results for $C = 0$ are quite different from the other ones, even though the average win percentage is about the same. Here there are four games that (almost) get a perfect score, which are Aliens, Butterflies, Chase and Missile Command. With other C parameter settings, only Aliens and Butterflies perform that well. Some of the other games (Frogs) perform less well than normal, and other perform about equal.

What can be concluded from these results? Firstly, setting the C parameter too high will result in bad performance. The algorithm needs a good amount of exploitation to be effective. Some tests were ran with even higher C values and these confirm the statement, as they have even lower win percentages. Secondly, we can see that the C parameter isn't very sensitive for this algorithm, as all values around 1 score about the same. Even $C = \frac{1}{\sqrt{2}}$ doesn't produce any notable differences. Lastly and most importantly, setting the C parameter to 0 works particularly well for some games, especially Chase and Missile Command score a lot better. As also stated in the previous Section 6.1, these are games that benefit from getting score boosts as fast as possible. Setting the C parameter to 0 stimulates the agent much more towards actions that get high score values. The tree now only grows in the direction of high scoring states, rather than states that might be high scoring in the future.

It might be beneficial for the algorithm to do something with the C parameter, so the games that perform well on the low C value take advantage of this, while there is still a good amount of exploration happening too. This will be tested, among other things, in the next Section 6.3.

6.3 Final Improvements To The Algorithm

Throughout the writing of this thesis and experimenting with the KB Fast-Evo algorithm, ideas for improvements came up that might increase performance of the algorithm even more. This section will show the implementation of the most successful ideas, in order to get the best performing implementation of the KB Fast-Evo algorithm currently possible. The improvements are based on the suggestions from Chapter 4 that weren't featured in Chapter 5, and are based on observations from watching the algorithm play the games.

The following four ideas were implemented and turned out to improve the algorithm separately and together. All will be explained in detail.

- Decisive moves. This technique is simple and proved successful before for the MCTS algorithm.

- $C = 0$. Section 6.2 has shown increase in performance when setting the C parameter to 0.
- Courage over time. The agent will be less 'scared' as the game progresses in order to get out of local optima.
- Knowledge factor. As the knowledge change (ΔZ) from the original algorithm did not work optimally, this factor will guide the agent to more rare events.

The first improvement, decisive moves, was also mentioned as a general improvement for the MCTS algorithm in Section 2.5.2. In this particular implementation each action at action selection is checked to see if it will lead to a game-over state. If so, it will get the lowest possible number of visits so it's never picked (action selection here is based on amount of visits). Also, each action will be checked for a win-state or score increase and if so, that action is immediately picked. This simple technique has a huge influence on performance. Mostly because it makes the agent die less because of collisions with NPCs (Frogs, Boulderdash), but also because the agent will surely win the game or increase score when next to a 'good' sprite (Frogs, Zelda, Chase).

As can be seen in previous Section 6.2, setting the C parameter to 0 in the original KB-Fast Evo algorithm caused a big increase in performance for some games (Chase, Missile Command), while other games performed worse because there was not enough exploration in the tree policy. With all improvements combined, setting the C parameter to 0 no longer caused a performance decrease for certain games, but still the beneficial effects for the other games where there. Especially the new evolutionary strategy and the knowledge factor are reasons why exploration in the game tree no longer is necessary. The agent will be attracted to events that haven't occurred often yet, and when they have occurred often, the agent knows the value increase even before the game tree is built, so exploring possible score increasing actions has become mostly irrelevant.

Courage over time was an improvement that resulted from observing the games a lot. Especially for games such as Frogs and Zelda it was clearly visible that the agent would often be stuck in the same corner of the map because there were 'bad' NPCs between the agent and the objective. The agent would find not dying more important than the score boost and thus would be stuck in a local optimum if no other easier paths could be found around the NPCs. This technique linearly increases negative score changes values (ΔR) in the scoring function over time. In the first time step of the game, the negative score change is as normal, in the final time step they are 0. This results in the agent behaving as normal at the start of the game, but as the game lasts longer and the agent has not yet been able to win it, the agent will take more and more risks until he *does* win (or lose) the game.

Finally, the knowledge factor was an idea that resulted from the analysis of the scoring function in Section 4.2.2.1. There, the observation was made that the knowledge change (ΔZ) did not add much towards the performance of the algorithm. The idea was good: the agent should be stimulated towards events that haven't occurred often yet in order to learn their value. The reason it performed not as expected was mostly because it applied only to events that did not result in a score change, because otherwise only the score change value (ΔR) is used, and because it converged to 0 very quickly. So now the knowledge factor K_i^F is introduced, which increases the distance change value for events that have rarely occurred. The agent is thus stimulated more towards rare events in order to quickly learn their value, or towards hard-to-reach events, such as the goal tile in Frogs or the key in Zelda. The following formula describes the knowledge factor K_i^F:

$$K_i^F = 1 + \frac{9Z_{totF} - 9Z_{iF}}{Z_{totF}Z_{iF} - Z_{iF}} \tag{6.1}$$

With the new scoring function then as follows:

$$\Delta D^K = \sum_{i=1}^{N} \Delta(D_i) \times K_i^F \tag{6.2}$$

$$Reward = \begin{cases} \Delta R & : \Delta R \neq 0 \\ \alpha \times \Delta Z + \beta \times \Delta D^K & : \text{Otherwise} \end{cases} \tag{6.3}$$

Z_{iF} is the number of occurrences of event i after the roll-out. Z_{totF} is the total amount of all event occurrences (in the game) after the roll-out. This results in a factor between 1 and 10, where a relatively low Z_{iF} will be closer to 10, and a higher Z_{iF} will quickly converge to 1. This factor is multiplied with the distance change, which thus will increase when the distance to more rare events has decreased.

The results of the final implementation of the improved KB Fast-Evo algorithm can be seen in Table 6.3.

The results show an average increase in performance of over 10% when compared to the original KB Fast-Evo algorithm. Also, 4 games get a 100% win percentage now, as opposed to just 1. Another thing that stands out is that the average amount of roll-outs is actually higher than in the original algorithm, so this implementation is faster. The only cause of this is setting the C parameter to 0. There are 3 more games that perform a lot better, but not optimal yet, which are Boulderdash, Frogs and Zelda. There is one game that performs worse: Survive Zombies.

Game	**Win %**	**Avg. score**	**Min. score**	**Max. score**	**Avg. roll-outs**
Aliens	100.0%	68.76 ± 13.33	43	82	98.72
Boulderdash	56.0%	22.76 ± 8.09	9	47	26.38
Butterflies	100.0%	26.08 ± 10.95	14	52	149.55
Chase	100.0%	9.88 ± 3.21	6	15	107.40
Frogs	52.0%	-0.28 ± 1.46	-2	1	58.09
Missile Command	100.0%	10.2 ± 4.20	5	16	161.94
Portals	24.0%	0.24 ± 0.44	0	1	145.69
Sokoban	16.0%	1.00 ± 0.91	0	3	153.04
Survive Zombies	28.0%	23.52 ± 17.14	0	58	48.69
Zelda	68.0%	2.40 ± 2.10	-1	6	109.06
Average	64.4%	16.46 ± 6.18			105.86

Table 6.3: Results of the final implementation of the KB Fast-Evo algorithm, including path finding, improved evolutionary strategy, courage, knowledge factor, decisive moves and $C = 0$.

The main reasons for the performance increases are that the agent now is much more eager and likely to get score boosts and is less likely to 'die'. The other very important reason is that the path finding, knowledge factor and courage over time will get the agent out of local optima much more frequently. The games that were hard for the agent throughout this thesis (Sokoban, Portals) are still hard for this implementation. Portals mainly because of the scoring system in the game (no points for going through a portal), so the agent rarely makes it far enough to see the goal. Sokoban is still hard because the agent has to push in a certain direction, which is not a part of the agent's search space and because the agent cannot search deep enough to solve the puzzles.

The one game that surprisingly performed worse is Survive Zombies. For a small part this has to do with courage over time, because towards the end the agent is a bit more likely to die to the zombies. But for the most part this has to do with the decisive moves technique. This was very unexpected, but can be explained as follows. With decisive moves, the agent will always go for a score boost when next to it, which in this case is either picking up honey, or killing a zombie *with* the honey. In the original implementation the agent would be able to see far enough ahead that he would run out of honey quickly when killing zombies and then dying to other nearby zombies when no more honey is in the inventory. In this implementation the agent can also see that he would die, but due to the decisive moves technique ignores that and kills a zombie anyway when next to it. This causes the agent to run out of honey very quickly and then dying when cornered by the zombies.

The algorithm has improved a lot overall when compared to the original KB Fast-Evo algorithm. The smaller and easier to implement techniques from this chapter increased performance a lot more than the big changes from Chapter 5. This was unexpected, but it was an interesting result nonetheless. Still there is room left for more improvement of this algorithm, but whether it can ever be fully optimal is questionable. A lot more discussion about this and a final conclusion can be found in the next Chapter 7.

Chapter 7

Discussion And Conclusion

The final chapter in this thesis will discuss the findings from the above chapters and will explain what can be learned from them. The research questions will then be answered. Lastly, some suggestions for future work will be given.

7.1 Discussion

Even though the KB Fast-Evo algorithm as described by Perez et al. [4] initially seems like a very different and good approach to use the MCTS algoritm for GVGP, some issues arose as can be seen in Chapter 4. KB Fast-Evo didn't improve upon the vanilla MCTS algorithm as much, mainly because the results of the vanilla MCTS algorithm were so much higher than in the original paper. Still the KB Fast-Evo has a significant higher average win-rate, so it definitely is a better algorithm. But this improvement is almost exclusively because of the distance change value in the scoring function and even that value was sub-optimal (euclidean distance). As could be seen, the evolutionary part hardly influences the roll-outs, mostly because of position-sensitive move action weights and the quick convergence of the matrix evolution. Also the knowledge change value in the scoring function yields no large enhancement to performance.

The issues with the KB Fast-Evo algorithm that were found, were handled in next chapter. But even though (most of) the issues mentioned were solved there, performance did not nearly increase as much as expected. What can be concluded from this? For the path finding part, it means that performance can only be increased slightly more if that algorithm is improved further, by letting the agent learn obstacles for instance. But only if that doesn't take much additional computation time. For the evolutionary part the question is harder to answer, because there are so many ways to approach this. If the parameters for the evolutionary algorithm are fine tuned some more, there will probably be a slight increase in performance, but most likely the evolutionary algorithm has to be *much* more sophisticated than it is now. This is a representational problem that might not be easily solved. Also, steering the roll-outs turned out to be less influential than first thought. In order to really improve the algorithm, a much deeper understanding of the games should be realized, rather than what 'good' and 'bad' sprites are. This could definitely also be done with an evolutionary approach, but just 'move' and 'use' weights to bias the roll-outs no longer suffices. Pattern recognition or some way to learn the goal of the game are suggestions for improvement.

Some additional smaller changes to the improved KB Fast-Evo algorithm were implemented in Chapter 6, which *did* turn out to have a big positive impact on performance. This is good news, because there might be room for even more improvement when new or existing MCTS techniques are applied to this algorithm. But, as also asked at the end of that Chapter 6, will the algorithm ever become fully optimal? On its own... probably not. MCTS just has some serious problems when applied to a game such as portals, where the scoring system only gives 1 point for a hard-to-reach goal and otherwise awards no other scores. A game like Sokoban is also very hard for this algorithm, as it is forced to solve a puzzle that requires looking many steps ahead. However, 4 out of 10 games got a perfect score, so that means the algorithm certainly is good in *some* games. The conclusion that could be drawn here is that the algorithm by itself isn't enough to solve the vastly different games GVGP provides, but combinations with other algorithm might result in great performance. Recognizing a puzzle-type game (like Sokoban) and then using a breadth-first search instead of MCTS is one example of combining algorithms.

7.2 Research questions

How does the KB Fast-Evo algorithm compare to a standard MCTS implementation for GVGP?

The most basic (vanilla) implementation of the MCTS algorithm already wins a substantial percentage of the games in the tests. Some games even get a perfect score, but others are almost never won. The KB Fast-Evo algorithm outperforms the vanilla MCTS implementation significantly overall, but because of some issues with the algorithm, on some games performance is equal or even worse than the vanilla MCTS implementation.

What improvements can be made to the KB Fast-Evo algorithm for GVGP?

There is a lot of room for improvement of the KB Fast-Evo algorithm. Especially in the scoring function and the way the roll-outs are biased. Here, a path finding algorithm was applied that results in more accurate values in the scoring function. Furthermore, the evolutionary part of the algorithm was changed a lot, which now produces a better weight matrix for guidance of the roll-outs.

What other techniques can be used to maximize performance of the MCTS algorithm for GVGP?

Fine-tuning parameters, such as the C parameter in the UCT formula, can increase performance significantly. Also existing improvements for the MCTS algorithm such as the decisive moves technique prove to be beneficial. Finally, techniques that can help get the agent out of local optima increases performance. For instance, in this thesis 'courage over time' was introduced for this purpose.

7.3 Future Work

Here a brief overview will be given with suggestions for future work. Most of these were already mentioned in this chapter or earlier in the thesis.

- Improving the path finding algorithm. This can be done with a more efficient algorithm than basic A* and also by letting the agent learn what obstacles are.
- Features and feature values can be researched more in-depth. There could be a feature value that performs better. Also, adding additional features such as collision direction might be beneficial.
- Further improvement of the scoring function. Here, the events caused by the avatar and its spawned sprites (bullets, swords, etc.) are taken together. These could be separated for possible enhanced performance.
- Fine-tuning, or even completely changing the evolutionary approach of the KB Fast-Evo algorithm
- Combining the MCTS algorithm with another algorithm to cover a broader range of problem spaces (games).

Appendix A

The Training Set In Detail

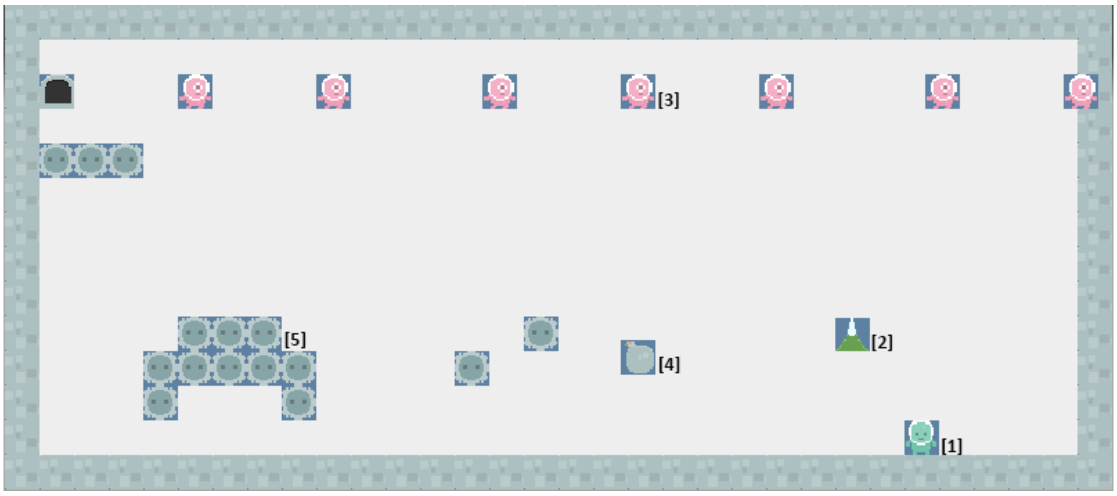

FIGURE A.1: **Aliens**
[**1**] The avatar. Can move from side to side and shoot upwards. Dies when an alien or bomb touches it.
[**2**] Avatar projectile. Moves up and destroys an alien on contact.
[**3**] Alien. Moves down in a zigzag pattern, sometimes drops bombs.
[**4**] Bomb. Falls down and kills the avatar on contact.
[**5**] Protective structure. Doesn't move and is destroyed when the projectile touches it.

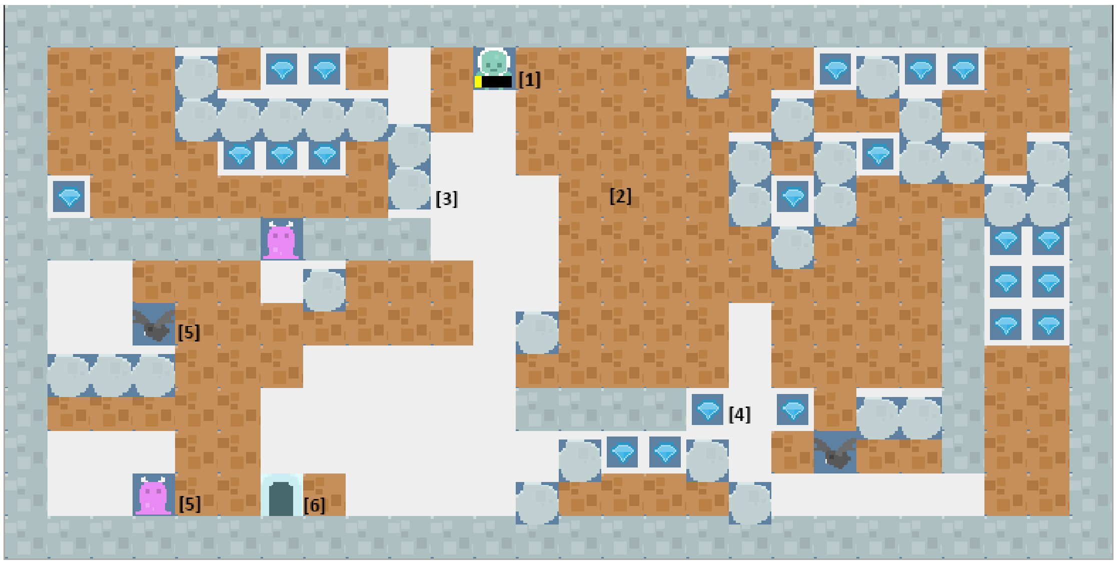

FIGURE A.2: **Boulderdash**
[1] The avatar. Can move in all directions and use a shovel to dig through the dirt.
[2] Dirt. Blocks the avatars path until it's removed with the shovel.
[3] Boulder. Blocks the avatars path. Cannot be removed but falls down when dirt is dug from under it. Kills the avatar if it falls on top of him.
[4] Diamond. The avatar can collect these. Gives points and is kept in the avatar's inventory.
[5] Enemies. Move around, but cannot go through the dirt. Kill the avatar on contact and spawn a new diamond if in collision with another enemy.
[6] Exit. Wins the game if the avatar reaches it with 10 diamonds in its inventory.

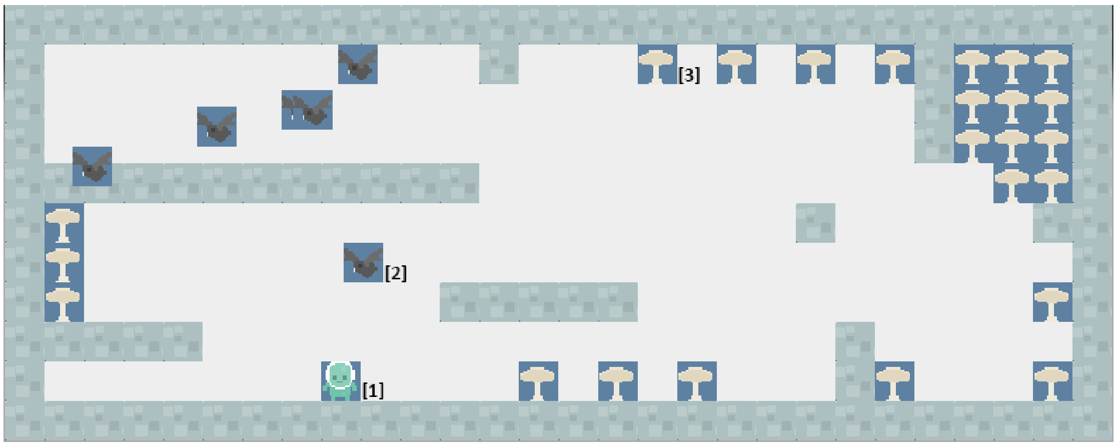

FIGURE A.3: **Butterflies**
[1] The avatar. Can move in all directions.
[2] Butterfly. Moves around randomly and is destroyed when the avatar collides with it. When all butterflies are destroyed the game is won.
[3] Cocoon. If a butterfly touches it, it is destroyed and a new butterfly is spawned. The game is lost when all cocoons are gone.

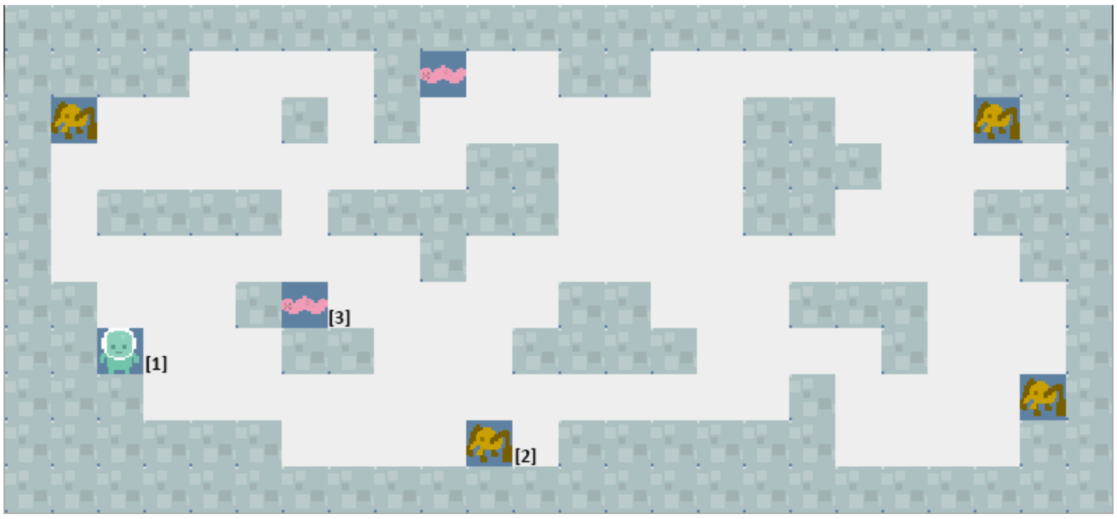

FIGURE A.4: **Chase**
[1] The avatar. Can move in all directions.
[2] Scared goat. Runs away from the avatar and is destroyed when the avatar touches it. When all scared goats are destroyed the game is won.
[3] Goat corpse. When a scared goat touches this, the scared goat turns angry and will chase and kill the avatar on contact.

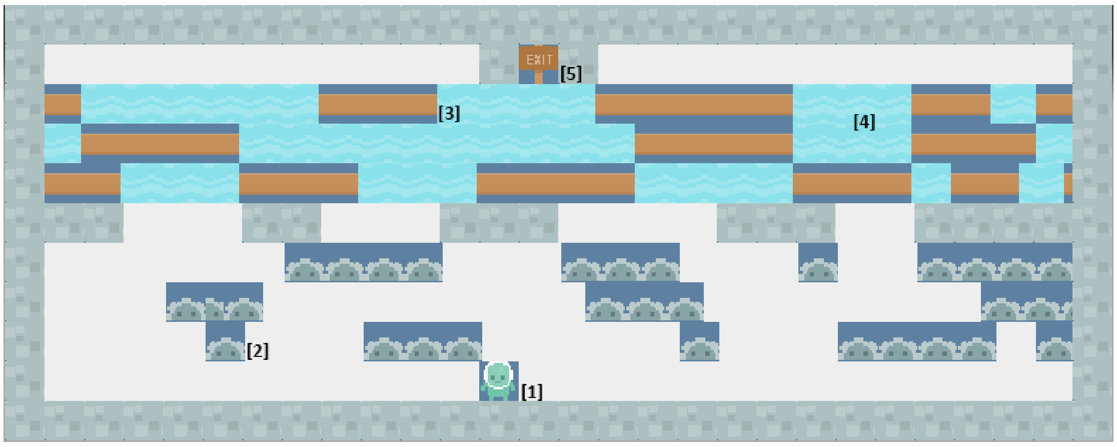

FIGURE A.5: **Frogs**
[1] The avatar. Can move in all directions.
[2] Truck. Moves from one side of the screen to the other. Kills the avatar on contact.
[3] Log. Moves from one side of the screen to the other. The avatar can move across this.
[4] Water. Kills the avatar on contact.
[5] The goal. If the avatar reaches this, the game is won.

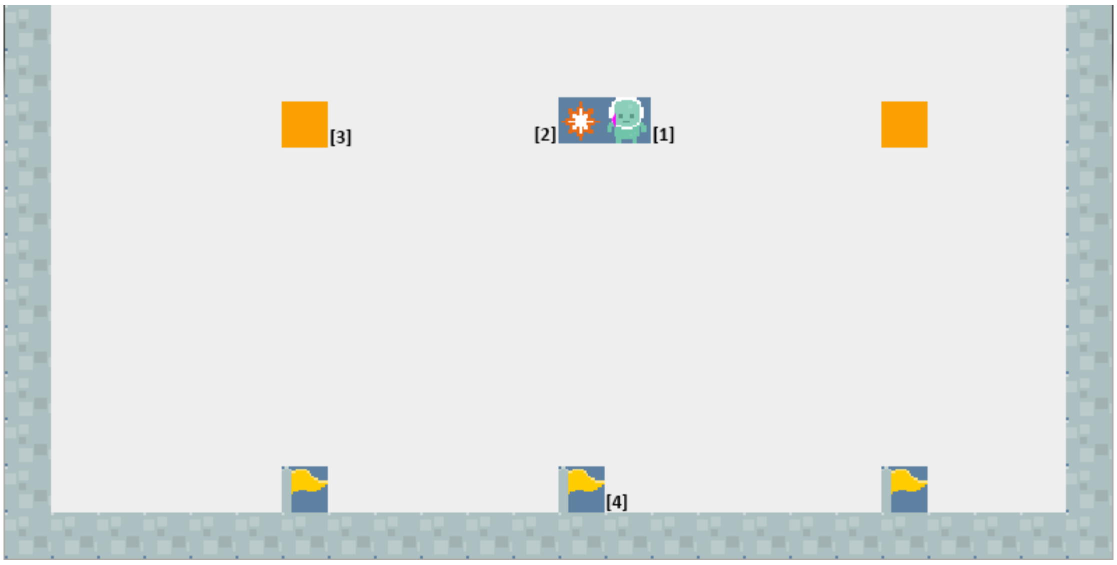

FIGURE A.6: **Missile Command**
[**1**] The avatar. Can move in all directions and spawn an explosion next to it.
[**2**] Explosion. Doesn't move and lasts a few time steps. Destroys a missile on contact.
[**3**] Missile. Rapidly moves towards a nearby city and destroys it on contact.
[**4**] City. Is destroyed when a missile touches it. When there are no more missiles and at least one city survives, the game is won.

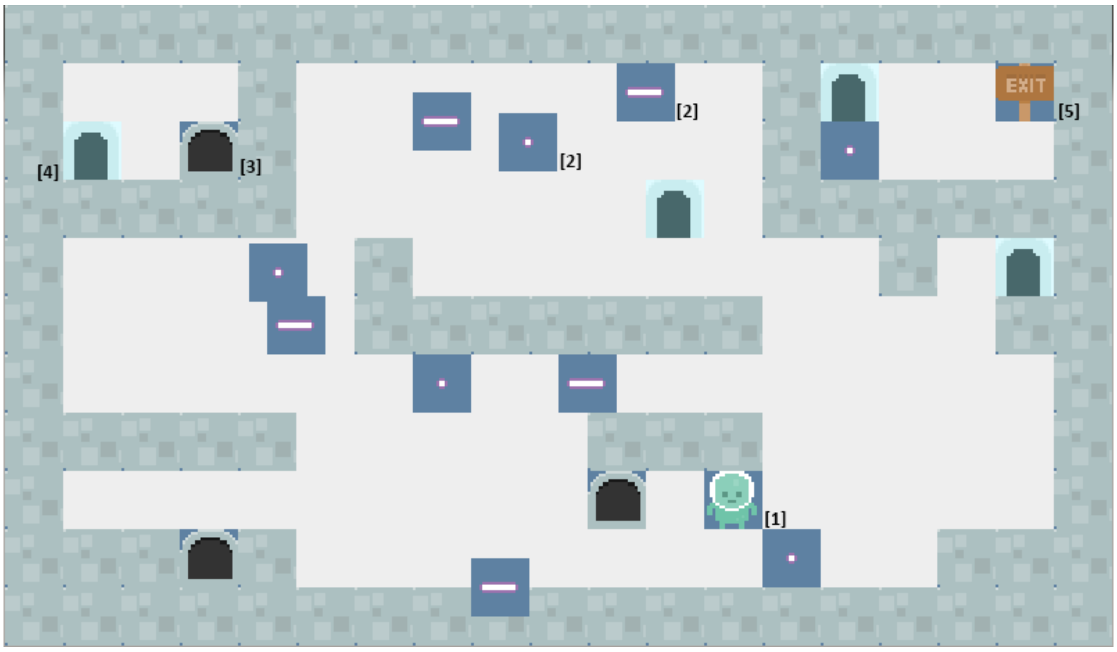

FIGURE A.7: **Portals**
[**1**] The avatar. Can move in all directions.
[**2**] Lasers. Move around in a pattern and kill the avatar on contact.
[**3**] Entrance door. Spawns the avatar at a certain exit door when the avatar reaches it.
[**4**] Exit door. Spawns the avatar when the appropriate entrance door was touched.
[**5**] The goal. When the avatar reaches this, the game is won.

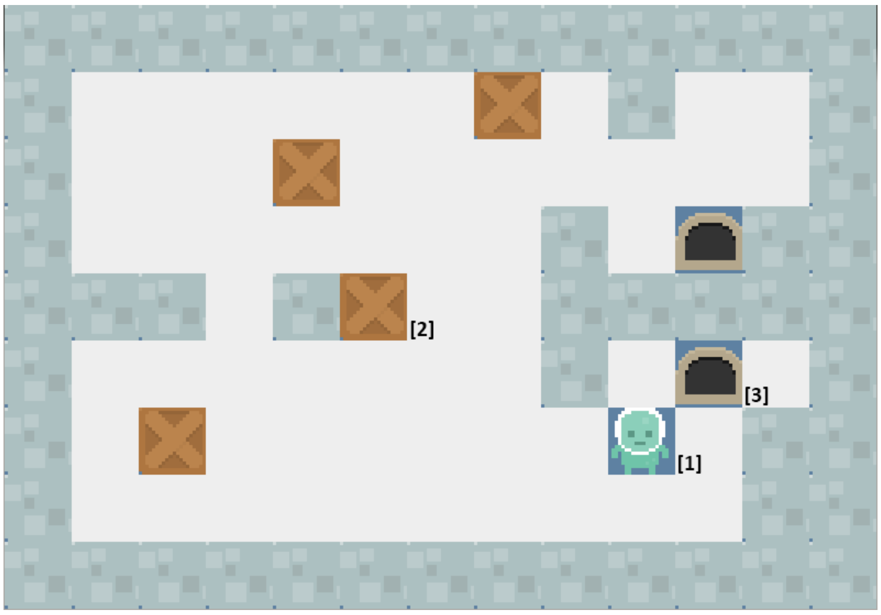

FIGURE A.8: **Sokoban**
[1] The avatar. Can move in all directions.
[2] Box. Move around in the direction the avatar pushes it. Disappears when it touches a hole. The game is won when all boxes are gone.
[3] Hole. Removes a box from the field when the box is pushed in to it.

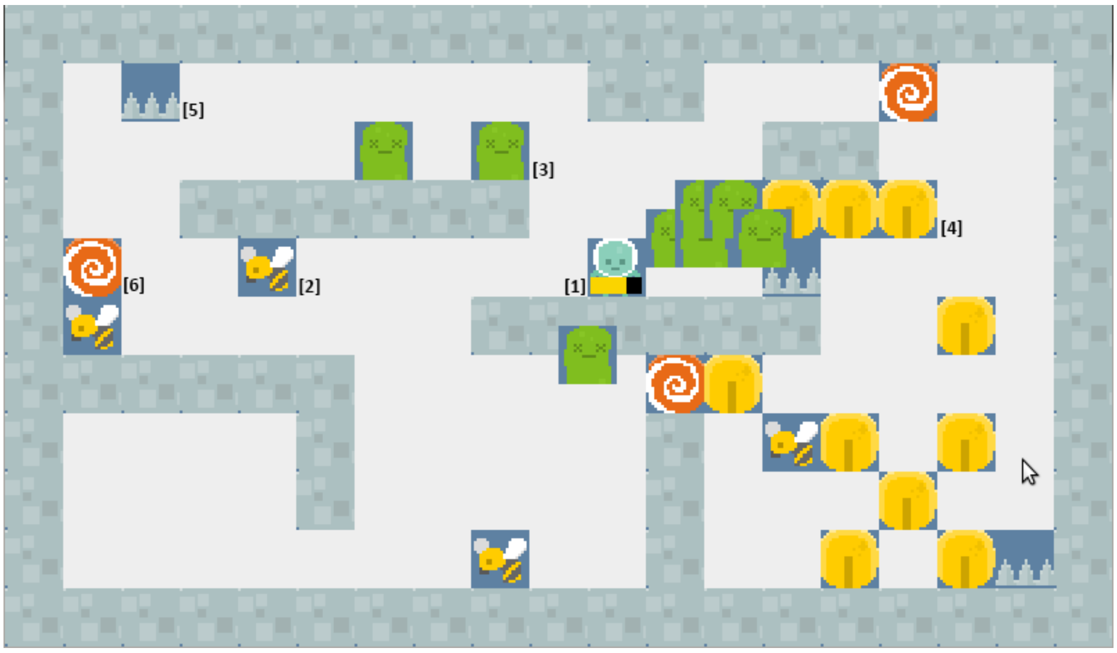

FIGURE A.9: **Survive Zombies**
[1] The avatar. Can move in all directions.
[2] Bee. When it touches a zombie, both sprites are destroyed and honey is spawned.
[3] Zombie. Is killed when the avatar touches it with honey in its inventory (one honey is consumed), but kills the avatar when no more honey is available.
[4] Honey. The avatar can collect this and use it to kill zombies.
[5] Grave. Occasionally spawns a zombie.
[6] Bee hive. Occasionally spawns a bee.

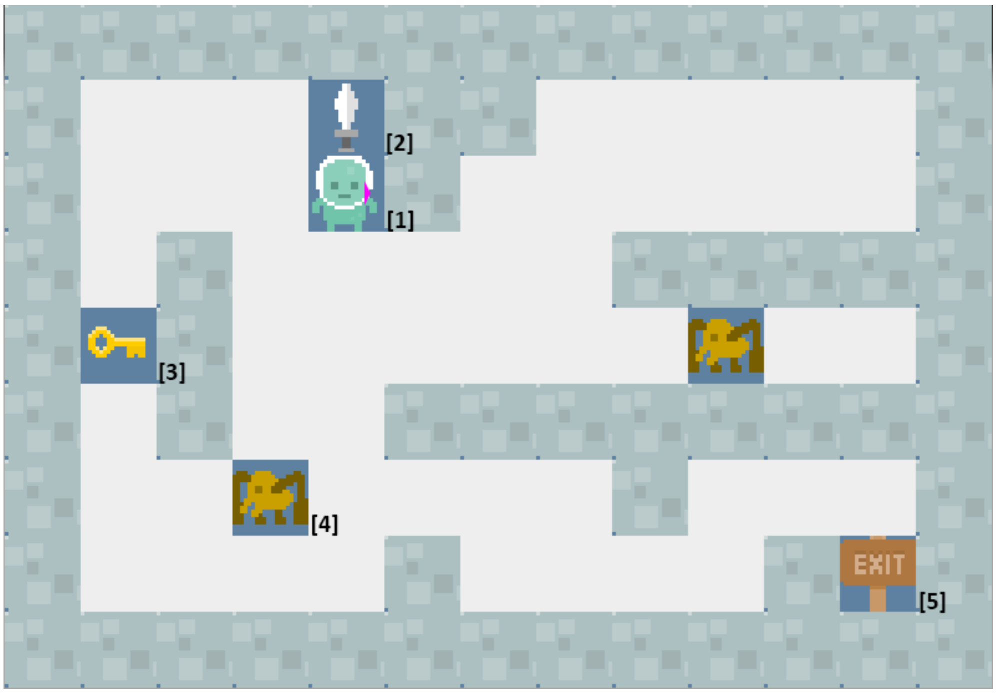

FIGURE A.10: **Zelda**
[**1**] The avatar. Can move in all directions and can use a sword to kill enemies.
[**2**] Sword. Kills an enemy when it touches the enemy.
[**3**] Key. The avatar must collect this before going to the goal.
[**4**] Enemy. Kills the avatar on collision with the avatar, but dies when it touches the avatar's sword.
[**5**] The goal. The game is won when the avatar reaches this with the key in its inventory.

Bibliography

[1] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. *5th International Conference, CG 2006*, pages 72–83, 2006.

[2] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. *17th European Conference on Machine Learning, Berlin, Germany*, pages 282–293, 2006.

[3] Guillaume Chaslot, Er Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A new framework for game AI. In *Proceedings of AIIDEC-08*, pages 216–217, 2008.

[4] Diego Pérez, Spyridon Samothrakis, and Simon M. Lucas. Knowledge-based fast evolutionary MCTS for general video game playing. *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2014.

[5] L. Kocsis, C. Szepesvári, and J. Willemson. Improved Monte-Carlo search. *Univ. Tartu, Estonia, Tech.*, 2006.

[6] G. M. J. Williams. Determining game quality through UCT tree shape analysis. *M.S. Thesis, Imperial College, London*, 2010.

[7] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, vol. 47:235–256, 2002.

[8] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. *Proc. Adv. Neur. Inform. Process. Syst., Vancouver, Canada*, 2006.

[9] P. Hingston and M. Masek. Experiments with Monte Carlo Othello. *IEEE Congress on Evolutionary Computation, Singapore*, pages 4059–4064, 2007.

[10] S. Samothrakis, D. Robles, and S. M. Lucas. A UCT agent for Tron: Initial investigations. *IEEE Symposium on Computational Intelligence and Games (CIG), Dublin, Ireland*, pages 365–371, 2010.

[11] F. Teytaud and O. Teytaud. On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. *IEEE Symposium on Computational Intelligence and Games (CIG), Dublin, Ireland*, pages 359–364, 2010.

[12] James H. Brodeur, Benjamin E. Childs, and Levente Kocsis. Transpositions and move groups in Monte Carlo Tree Search. *IEEE Symposium on Computational Intelligence and Games (CIG), Perth, WA*, pages 389–395, 2008.

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

[14] Y. Soejima, A. Kishimoto, and O. Watanabe. Evaluating root parallelization in Go. *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 4: 278–287, 2010.

[15] A. Bourki, G. M. J.-B. Chaslot, M. Coulm, V. Danjean, H. Doghmen, J.-B. Hoock, T. Hérault, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu. Scalability and parallelization of Monte-Carlo Tree Search. *7th International Conference, CG 2010, Kanazawa, Japan*, pages 48–58, 2010.

[16] J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11:1203–1212, 1989.

[17] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. *Proceedings of the 24th international conference on Machine learning, Corvalis, Oregon*, pages 273–280, 2007.

[18] Guillaume Chaslot, Mark Winands, Jaap H. van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo Tree Search. *Joint Conference on Information Sciences, Heuristic Search and Computer Game Playing Session*, 2007.

[19] Y. Björnsson and H. Finnsson. CadiaPlayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1:4–15, 2009.

[20] Y. Björnsson and H. Finnsson. Learning simulation control in general game-playing agents. *24th AAAI Conference on Artificial Intelligence, Atlanta, Georgia*, pages 954–959, 2010.

[21] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, vol. 3:9–44, 1988.

[22] S. Gelly and D. Silver. *Artificial Intelligence*, vol. 175, no. 11.

[23] D. P. Helmbold and A. Parker-Wood. All-Moves-As-First heuristics in Monte-Carlo Go. *International Conference on Artificial Intelligence (ICAI), Las Vegas, Nevada*, pages 605–610, 2009.

[24] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. Genetic Programming: An introduction. *The Morgan Kaufmann Series in Artificial Intelligence*, 1997.

[25] Tristan Cazenave. Evolving Monte-Carlo Tree Search algorithms. *Dept. Inf., Univ. Paris*, vol. 8, 2007.

[26] James Pettit and David Helmbold. Evolutionary learning of policies for MCTS simulations. *Proceedings of the international conference on the Foundations of Digital Games, New York, USA*, pages 212–219, 2012.

[27] A. Benbassat and M. Sipper. EvoMCTS: Enhancing MCTS-based players through Genetic Programming. *IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[28] Atif M. Alhejali and Simon M. Lucas. Using Genetic Programming to evolve heuristics for a Monte Carlo Tree Search Ms Pac-Man agent. *IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[29] Simon M. Lucas, Spyridon Samothrakis, and Diego Pérez. Fast evolutionary adaptation for Monte Carlo Tree Search. *17th European Conference, EvoApplications 2014, Granada, Spain*, pages 349–360, 2014.

[30] Michael Genesereth and Nathaniel Love. General game playing: Overview of the AAAI competition. *AI Magazine*, vol. 26:62–72, 2005.

[31] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. *Technical Report LG200601, Stanford University*, 2006.

[32] John Levine, Clare Bates Congdon, Michal Bida, Marc Ebner, Graham Kendall, Simon Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General video game playing. *Dagstuhl Follow-up*, vol. 6, 2013.

[33] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon Lucas. The general video game AI competition. www.gvgai.net, 2014.

[34] Joel Lehman and Risto Miikkulainen. General video game playing as a benchmark for human-competitive AI. In *AAAI-15 Workshop on Beyond the Turing Test*, 2015.

[35] Marc Ebner, John Levine, Simon Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. *Dagstuhl Follow-up*, vol. 6, 2013.

[36] Tom Schaul. A video game description language for model-based or interactive learning. *IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[37] Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. *12th International Conference on Advances in Computer Games (ACG), Pamplona, Spain*, pages 1–13, 2009.

[38] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2:251–258, 2010.