MCTS BASED AGENT
FOR
GENERAL VIDEO GAMES

Torsten Schuster

Master Thesis DKE 15-14

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
July 8, 2015

Abstract

Games are commonly used in different fields of Artificial Intelligence to benchmark techniques. In many cases, the developed agents focus on playing a single game and apply domain-specific knowledge to improve their results. The General Video Game AI Competition is a competition that addresses the challenge of developing agents for a variety of games. The competition uses adaptations of popular games, such as *Zelda* or *Pac-Man*, as well as newly invented ones to stimulate research in general game playing. The challenges, goals and rules of the competition are described as well as the underlying framework. Aspects of the framework, which are important for the development of an agent, such as important classes and game characteristics, are introduced. The non-deterministic character of games, the time limitation to 40 milliseconds for executing an action as well as the low performance of the framework are identified as major challenges.

Monte-Carlo Tree Search (MCTS) in its basic form does not require domain knowledge and can therefore be applied to general game playing. Agents that are based on MCTS performed well in the competition of 2014 and can be enhanced in several ways. This thesis explains and applies existing techniques for enhancing an MCTS based agent to perform well in the GVG-AI Competition. Amongst these enhancements are common ones, for instance Upper Confidence Bounds for Trees, to influence the selection, and more complex ones, such as Evolutionary Algorithms to guide the play-out. The developed agent is designed to combine different strategies and enhancements. The results of applying strategies on their own as well as combinations are evaluated in the context of the GVG-AI Competition. Some strategies show only significant differences in performance between specific games. Other strategies improve the overall performance significantly when used on their own. Combinations of promising enhancements performed well, but without significant improvement in the overall performance. The resulting agent developed for this thesis performed better than the third ranked SAMPLE MCTS PLAYER of the GVG-AI Competition in 2014 in most tests.

Contents

Chapter 1

Introduction

Games play an important role in the development and benchmarking of Artificial Intelligence. This chapter introduces different types of games and solutions as a background for this thesis. It also gives an overview of all following chapters.

Chapter contents: Games in Artificial Intelligence, Techniques for Playing Games, Problem Statement & Research Questions and Outline

1.1 Games in Artificial Intelligence

Different types of games have been playing an important role in Artificial Intelligence (AI). Popular examples are board games like chess or Go, which are used to benchmark AI for games. Chess programs improved to a degree on which they can compete with world champions or even defeat them (Hsu, 2002). Go experts can still outperform programs, especially on larger boards, but the agents have improved their results for the past years (Clark and Storkey, 2014).

Other types of games face different challenges and use different techniques related to AI. IBM's *Watson*, for instance, won against human experts in the quiz show *Jeopardy!* using, for instance, natural language processing and information retrieval (Ferrucci *et al.*, 2013). Most existing agents for playing these games are developed specifically for this particular game, like chess, Go or *Jeopardy!*. These agents often use domain-specific knowledge or data bases to perform the best possible action. The result is, in many cases, a highly specialized agent that performs well in one specific domain, but cannot be transferred easily to a different problem. The idea of general game playing is to design a program that is able to perform well in many different types of games (Levine *et al.*, 2013). Therefore, no or not much domain knowledge is used. Nevertheless, techniques that were used for domain-specific tasks, such as playing *Reversi* (Benbassat and Sipper, 2014) or *Ms. Pac-Man* (Robles and Lucas, 2009), can be used and extended.

General game playing has become more popular during the last years. So far, there were several competitions at AI Conferences, like the annual General Game Playing competition at the AAAI Conference (Decoster, 2014) which takes place since 2005. The competition focuses on abstract games, such as chess or Go. This thesis concentrates on general video games in the context of the General Video Game AI Competition. The video games used for the competition have some other challenges like time constraints due to the real time environment. The competition uses adaptations of popular games, such as *Zelda*, *Whack-A-Mole* or *Pac-Man*, as well as newly invented ones to stimulate research in general game playing.

Improvements in general video games can be beneficial for different domains. Techniques for an agent to autonomously learn how to play a game can be used to solve problems where no domain knowledge is available. This can be helpful for many applications like training autonomous systems. It is even possible to use these techniques to learn rules for games or other domains. These rules can then be used for data mining or machine learning.

1.2 Techniques for Playing Games

Games are commonly used in different fields of AI to benchmark techniques for solving problems. Search techniques and machine learning are two of the most common approaches for playing these games. To summarize, search techniques try to investigate multiple possible scenarios and find the most promising one. The goal of machine learning is to learn a model to play a game from existing data, like games played by human experts.

Both approaches, machine learning and search, can be divided into online and offline methods. Online methods update their model whenever new data arrives. Therefore, the model can be updated while the program is running. This is especially useful in a real-time environment where time or other resources for computing are limited. Offline methods have access to a whole data set for training. Thus, building the model usually takes more resources, but can deduce patterns using the whole data set.

The main concept of search based approaches is to investigate a search space based on the current state of the game. A common representation of a game is a tree. Each game state can be represented by a node and each action that transfers from one state to another state is represented by an edge. Starting at the current state of the game as root node, new levels of nodes for each player can be added. After the tree has grown to a certain size, for example after a certain amount of time elapsed or when all paths of a game are over, the most promising move can be chosen. During this process, the search techniques can explore the game by choosing unknown moves or exploit the game by choosing the potentially best move. Different search techniques use enhancements, such as heuristics, to guide this process or increase performance. Commonly used algorithms are alpha-beta pruning (Knuth and Moore, 1975) which tries to minimize the search space or Monte-Carlo Tree Search (MCTS) (Coulom, 2007) that tries to find the best move based on many random simulations. During the GVG-AI Competition in 2014, a basic sample agent that uses MCTS with common enhancements was ranked third. Amongst the winners, several agents were based on MCTS (Perez *et al.*, 2015). Thus, this thesis introduces MCTS as a promising approach for the GVG-AI competition with adaptations to the basic algorithm to increase performance. One example of these improvements is the use of machine learning to guide the random play-out in MCTS.

Machine learning has many applications such as object recognition, robotics or game playing. Depending on the technique, the given data set has to be labeled or unlabeled. A label for a game in a data set of games could be whether a game is won or lost. Techniques that use labeled data can be categorized as supervised learners, whereas techniques that use unlabeled data can be categorized as unsupervised learners (Han, Kamber, and Pei, 2011). There are other categories, such as semi-supervised learning or Reinforcement Learning. Using these different types of data sets, the techniques can build a model that can learn and improve a certain task. For example, a robot can learn to balance a pole or a computer can learn to play a game. In this thesis, Genetic Programming is investigated as one of the machine-learning techniques to guide the play-out in MCTS.

In most cases, the approaches of search techniques and machine learning are applied to one specific problem and in many cases, domain knowledge is used. This approach is not always applicable, as the domain can be unknown or no domain knowledge is available at all. This thesis introduces and improves an agent that uses a more general approach of playing games and tries to improve and combine existing techniques. To create the MCTS agent, methods from existing, domain-specific agents are used and applied to a more general agent. The main focus is on enhancing the selection and play-out step of MCTS. Existing approaches use Upper Confidence bounds for Trees (UCT) (Kocsis and Szepesvári, 2006) to improve the selection step or genetic algorithms to guide search with the use of learned features during the play-out (Perez, Samothrakis, and Lucas, 2014).

The idea of using Genetic Programming to enhance MCTS is not new. Benbassat and Sipper (2014) use strongly typed genetic programming (Montana, 1995) to learn an evaluation function during MCTS' play-out and use it to guide the play-out process. Perez *et al.* (2014) apply a similar approach to the General Video Game AI Competition and get promising results that can still be improved. Furthermore, there are many variations and improvements for MCTS and Genetic Programming. The combination, improvement and evaluation of these different approaches for general video games are the main focus of the thesis.

1.3 Problem Statement & Research Questions

As mentioned in the previous section, MCTS has been successfully used in AI for the past years but mostly for playing specific games like Go. Most existing AI for games use domain-specific knowledge to improve their results. Sometimes even human experts are necessary to label data to train agents. A basic MCTS agent does not need domain knowledge to play a game, but the introduction of domain knowledge may improve the performance. Some existing approaches of using MCTS for General Game Playing have been already promising (Finnsson and Björnsson, 2008; Benbassat and Sipper, 2014), but can still be improved. This leads to the following problem statement.

Problem Statement: *Which techniques for enhancing an MCTS based agent improves its performance in the GVG-AI Competition?*

The developed agent is able to play games without prior knowledge of rules or game characteristics. This problem statement covers different aspects of AI, such as General Game Playing or machine learning to be able to enhance the agent. The following four Research Questions (RQ) narrow and state the domain more precisely.

Research Questions:

RQ 1: Which existing approaches can be used to improve MCTS without using domain knowledge?

RQ 2: Which combination of techniques can be used to improve the performance of MCTS best in general video games?

RQ 3: What are the strengths and weaknesses of the different approaches for general video games and how can they be enhanced?

RQ 4: How well does the agent perform under different constraints?

To answer RQ 1 and RQ 2, different enhancements for MCTS are explained, evaluated and combined with other approaches, like Genetic Programming. The results of the evaluation lead to strengths and weaknesses mentioned in RQ 3. To find improvements, different characteristics of games and aspects of applied techniques have to be considered. In the field of general game playing, there are overall challenges like the absence of domain knowledge. The GVG-AI Competition adds additional constraints, such as a limited CPU time to choose an action. The answers to RQ 4 treat these aspects.

1.4 Outline

This thesis introduces existing approaches to play games and applies them to general video games. A major part for this thesis is the improvement of a General Game Playing agent for the General Video Game AI Competition. General Game Playing focuses on the development of general intelligences that can play more than one specific game (Levine *et al.*, 2013). Chapter 2 describes the resulting challenges and differences to other disciplines in AI. The General Video Game AI Competition is introduced as an example of a competition that is designed to face these problems. The goals of the competition are described, as well as the rules and the accompanying framework, that provides games for testing and an environment to run these games. This framework is fundamental for developing, testing and evaluating the agent.

The agent combines two existing approaches and applies them to General Game Playing. The first approach is MCTS, a tree search algorithm that is described in Chapter 3. MCTS can be divided into four steps, selection, expansion, play-out and backpropagation. Each step of the algorithm is explained as well as existing enhancements. The second approach is a special type of genetic algorithm that is explained in Chapter 4. This chapter summarizes the basic concepts of Genetic Programming and explains a modification, the so-called *Strongly Typed Genetic Programming.* The basic idea is to learn an evaluation function for a game state during the play-out of MCTS. The goal is to use this evaluation function in order to guide the play-out after some iterations. The genetic algorithm is explained in detail as well as the methods to incorporate it into MCTS.

To test the performance of the presented agent, several aspects have to be evaluated. Some of these are common features of General Game Playing, like handling completely different games with sometimes complex rules. Others are introduced by the General Video Game AI Competition, such as additional time constraints. Chapter 5 explains the test setup that is used to evaluate the agent's performance. Multiple configurations and combinations of the implemented improvements for MCTS and the genetic algorithm are evaluated. The results are compared to sample agents that are included in the framework and the results of the competition in 2014.

To summarize, this thesis uses existing approaches and modifies them to be applicable for the General Video Game AI Competition. As the competition is still new, there remain several aspects of improvement for the agents. Chapter 6 concludes the thesis and hints at these future research possibilities.

Chapter 2

General Video Game AI Competition

The General Video Game AI Competition (GVG-AI Competition) (Perez, 2015) is a competition that addresses the problem of creating a Artificial General Intelligence. Artificial General Intelligence should be able to perform well in multiple games. This chapter introduces the GVG-AI Competition with its goals and rules. To give a better understanding for the development of an agent participating in the competition, the provided framework is presented.

Chapter contents: Challenges & Goals, Competition & Rules, Framework and Game Characteristics

2.1 Challenges & Goals

Many agents in AI that play games focus on performing well in a specific game. Creating an Artificial General Intelligence that is able to play different games has additional challenges. DEEP BLUE, the chess playing agent that defeated the human world champion Garry Kasparov in 1997, for example focuses exclusively on chess. Although DEEP BLUE performed well in chess due to domain-specific knowledge, it would not be able to play another game, like Go. In chess, it is necessary to include domain knowledge to perform well. This kind of knowledge is not available in all domains. Other domains require work of human experts, which can be expensive, to acquire this knowledge.

Several competitions at AI conferences address these challenges. One of these competitions is the GVG-AI Competition that focuses on General Video Game Playing (Levine *et al.*, 2013). Section 2.2 explains the rules of the competition and gives additional background. The underlying framework for creating an agent for the competition is presented in Section 2.3.

The competition uses the competitive situation to stimulate research in the field of general game playing. Although the agents are not simultaneously playing against each other, the ranking in the competition is a good motivation. To benchmark the submitted agents, different games are used. Some games were developed for the competition, others are adaptations of existing games like Space Invaders, *Pac-Man* or *Zelda*. This combination of completely new as well as popular games is challenging and attractive.

The competition provides a framework that contains all necessary components to participate in the competition. This framework, for example, provides methods to evaluate a state, get the positions of game objects or simulate a move, so that the development of the agent can focus on AI techniques and not the analysis of the game's components. Participating in other competition tracks can, for example, require a visual analysis of the game. In comparison to other competitions, such as the General Game Playing competition (Decoster, 2014), the GVG-AI Competition does not provide information about the rules of the game, so that the agent cannot exploit them. Thus, the agent has to learn rules for choosing an action based on observations he has made in the game and a forward model that can be accessed to simulate moves. The forward model is explained in Subsection 2.3.1. To test an agent, different sets of games are provided. During the tournament a completely new set of games is used. This reduces the danger of overfitting an agent as the agents cannot be specialized in a specific set of games.

Other challenges are a result of additional rules in the competition described in Section 2.2. A limitation of time for choosing an action, for example, results in the need of components for time management. All these challenges have impact on the development of an agent for the competition. This thesis presents an MCTS based agent, as MCTS is able to be adapted to these constraints, such as limited time. Even when the search technique is interrupted, MCTS is able to take a reasonable action. Furthermore, MCTS combined with other techniques, is able to play games without prior domain knowledge. These techniques increase the performance of the agent, using the information gained during multiple play-outs.

2.2 Competition & Rules

Competitions are a common approach to motivate research in a certain field of AI. Popular examples are chess championships or the RoboCup (The Robocup Federation, 2015), a robotics competition with different domains, such as soccer. Likewise, there are competitions in the field of general game playing, such as since 2005 the annual General Game Playing competition at the AAAI Conference (Decoster, 2014) or the GVG-AI Competition. The latter was held for the first time in 2014 at the IEEE Conference on Computational Intelligence and Games (IEEE CIG, 2015). In 2015 it will be held at the Genetic and Evolutionary Computation Conference (GECCO, 2015) and the CIG.

The two mentioned competitions differ in multiple aspects. The General Game Playing Competition (GGP Competition) benchmarks the competitors with different abstract games, such as chess, Tic-Tac-Toe or Chinese Checkers, whereas the GVG-AI Competition uses video games. Video games have different characteristics and rules that have influence on an agent which plays the games. For example, the GVG-AI Competition is executed in real time. An agent has only 40 milliseconds per game cycle to choose an action. If the time is exceeded by additional ten milliseconds, the agent is disqualified. Before a game starts, the agent has one second time for initialization. The rule when a game has to end is defined by each game itself. However, there is an additional global limitation to a maximum of 2000 game cycles. If the agent cannot end the game earlier, the game is lost. Other parameters define aspects like the maximum time a game can take.

During the competition, each agent will play ten unknown games with five levels per game. Each game is played ten times, so that each agent plays $10 \cdot 5 \cdot 10 = 500$ games in the tournament. The results of all played games are taken into account for the ranking of agents during the competition. The games are played by a single agent and can have multiple non-player characters (NPCs). Additional characteristics of the different games are described in Subsection 2.3.2.

The winner of the GVG-AI Competition is determined by three different aspects:

1. The highest number of won games
2. A score that is provided by the framework obtained by playing the game
3. Least time taken to play the game

According to theses aspects, an agent is ranked for each game. The agent with the highest number of victories is ranked first. In case of a draw, the score will be taken into account and the time for playing the game is used as last tie-breaker rule. Following the scoring system of Formula 1 (Formula One World Championship Limited, 2015), the agents get points for each game. The agent that gets the most points wins the competition.

Additional tracks with different focuses are planned for the GVG-AI Competition. The previously described rules are transferred to the so-called “Planning Track” (Perez, 2015) or “Gameplay Track” (Perez *et al.*, 2015). In addition, two more tracks are in development, the “Learning Track” and the “Procedural Content Generation Track”. The “Learning Track” does not provide a forward model, but agents get additional time to learn game characteristics. The “Procedural Content Generation Track” focuses on the generation of a valid level given a description of a game. Currently, the “Planning Track” is the only track where agents can be developed for. Thus, this thesis focuses on AI techniques for this track.

The rules and parameters of the competition have to be taken into account while developing an agent for the competition. Especially aspects like the limited amount of time per turn can be challenging.

The following section gives an overview of the provided framework to get a more detailed insight in the possibilities to create an agent.

2.3 Framework

2.3.1 Java Environment

The GVG-AI Competition provides a Java framework that consists of several packages. These packages include all classes necessary to run games and create new agents. This subsection introduces the core components of the framework as a background for creating and testing the MCTS based agent.

To be able to design versatile games for the competition, a Video Game Description Language (VGDL) (Ebner *et al.*, 2013) to describe these games is necessary. The framework for the GVG-AI Competition contains a Java version of PyVGDL Schaul (2014), which is written in Python. The Java port of PyVGDL is able to read text files with a definition of a game in VGDL and compiles it to be able to run in the Java environment. For each of these game representations, different levels can be designed. These descriptions can then be used to create a large variety of games with two dimensional levels for the competition. Subsection 2.3.2 presents some of the existing games used for the competition and their description.

The framework contains different classes and parsers to read game descriptions and represent the required game objects. These classes define properties of a game object, such as physical properties like mass or color. Other classes are used to control the game, for example to check if a game is over, or are base classes for developing custom agents. Some included sample agents are described in Subsection 2.3.3.

Additional resources, such as pictures to represent game objects, so-called sprites, can be used to visualize a game. Figure 2.1 shows three of these sprites that can be used for different game objects. The avatar sprite is used in most games for the agent's representation inside a game. The key is a passive object that is likely to be collected, but the sprite does not necessarily give information whether an object is harmful or friendly. For example, the ghost could be an ally or an enemy, depending on the game.

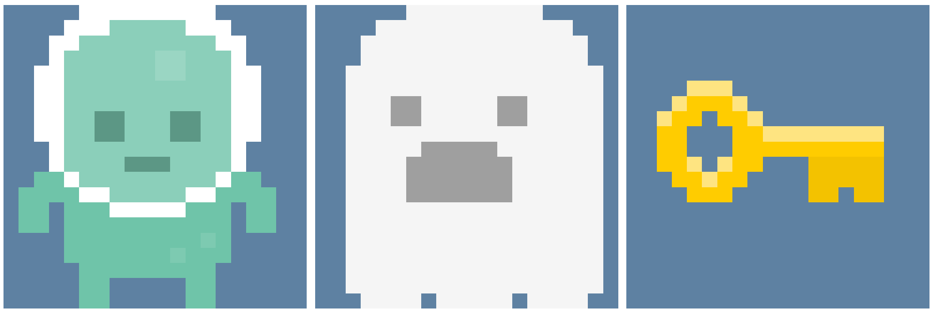

(a) Avatar (b) Ghost (c) Key

Figure 2.1: Example of sprites

Unlike the GGP Competition, an agent in the GVG-AI Competition has no access to the whole game description. Therefore, the agent has no direct access to the rules and characteristics of a game, such as the scoring system or the goal. Thus, analyzing observations and performing simulations is important to win the game. The remainder of this subsection focuses on the classes that are important while playing a game to obtain additional information. To start developing an agent, the framework contains an `AbstractPlayer` class. After creating a class that inherits from the `AbstractPlayer`, two methods have to be implemented. The first method is a constructor that is called whenever a new game starts. Thereby, the agent has time for initialization at the beginning of each game. The second method is an **`act`** method that is called every game cycle to retrieve an action from the agent.

Both methods are called with two parameters, a `StateObservation` and a `ElapsedCpuTimer` which provide information about the current state of the game and the remaining time. Table 2.1 describes these and other important classes of the framework. Using information provided by the different classes, such as the `Observation` and the `StateObservation`, the agent is able to decide which action to perform. The `StateObservation` is of particular importance as it contains a forward model that can be used to simulate future moves. The forward model allows to copy a state of the game and advance by executing an action. These methods are of particular importance for the MCTS based agent, as the play-out, described in Subsection 3.2.3, can use them.

Class	**Description**
`AbstractPlayer`	Base class for creating agents that are able to play the game. It requires a constructor and an `act` method that both get a `StateObservation` and an `ElapsedCpuTimer` as parameters.
`ElapsedCpuTimer`	Class that provides information about the elapsed time. It can, for example, be used in the agent's `act` method and constructor to choose an action to get information about the remaining time.
`Event`	Class to represent information about an event. An event is a collision between two objects, such as a sword and an enemy or the avatar and an enemy. It contains different properties, like the time when the event happened, the position and the participating game objects.
`Observation`	Class to provide information about different game objects in the game. An observation contains information about the type, such as a certain type of NPC or resource, of the observed object and its position.
`StateObservation`	Contains information about the state of a game, such as a score, available legal actions and observations. The actions are represented by `enum ACTIONS` and the observations by the `Observation` class. Furthermore, it wraps a forward model that can be used to simulate future moves and keeps a history of events represented by the `Event` class. The forward model allows to advance a state by applying an action. The result is one possible next state of the game, that does not have to be the outcome of applying this action in the real game due to the possible non-deterministic characteristics of a game.
`enum ACTIONS`	Enumeration that contains all actions that are available for all games. Not all actions have to be available in a game. *ACTION_UP*, *ACTION_DOWN*, *ACTION_LEFT* and *ACTION_RIGHT* can be used to move the avatar. *ACTION_USE* performs an interaction defined by the game. The use action in space invaders, for example, is to shoot or in *Zelda* to use the sword. *ACTION_NIL* means that no action is applied. If an agent does not choose an action in time, but before he gets disqualified, the framework assigns the *ACTION_NIL* for the current game cycle. *ACTION_ESCAPE* can be performed only by human players to quit the game.
`enum WINNER`	Enumeration that defines all possible outcomes of a game. The self-explaining types are *PLAYER_DISQ*, *NO_WINNER*, *PLAYER_LOSES* and *PLAYER_WINS*.

Table 2.1: Important classes and enumerations for the GVG-AI Competition

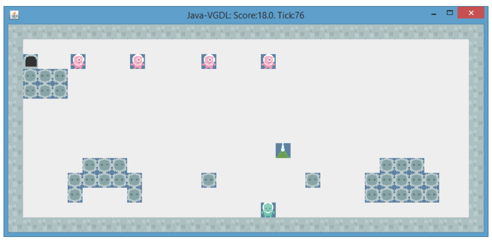
Games can contain certain random components, such as NPCs that perform random moves every game cycle. Executing an action by using the forward model follows the same rules as executing the action at the current state of the game. Thus, applying one action starting at the same game state can result in different outcomes due to the random components. In particular, simulating an action does not necessarily result in the same outcome as performing it. Still, it is possible to use the presented classes to get information about a game and its characteristics. For example, the forward model can be used to simulate actions multiple times to learn rules. Keeping track of different statistics can help to find a promising action. One common technique is to use MCTS with different enhancements to simulate multiple games as described in Chapter 3.

2.3.2 Game Sets

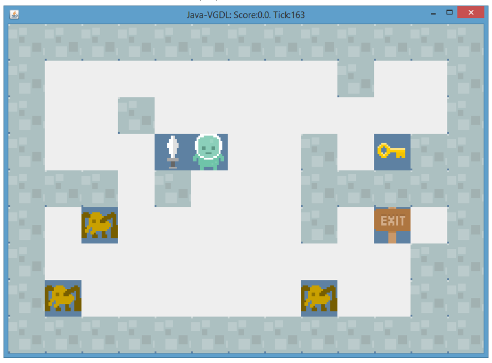
As the GVG-AI Competition aims at improving techniques for general game playing, the games of the competition have to be versatile. The previous section already introduced the VGDL that allows to describe a large variety of games with different characteristics. This section introduces some of the existing games used for the competition in 2014 and additional information about the description of the game.

The description of a game can be stored in a text file that consists of four blocks. The first block describes properties of different game objects, such as the agent's avatar or NPCs. These properties define physical properties of an object, such as color, shape, mass or momentum. Depending on the type of game object, additional properties, such as health, mana, ammunition or available actions can be defined. The second block describes interactions between two game objects. These interactions allow, for example, to describe the behavior, that an agent cannot walk through walls or that an agent can move boxes. The third block defines termination conditions, such as a win after all enemies are killed or a loss when the agent dies. The fourth block defines a mapping between game objects defined in the first block and a representation for level files. This allows to create multiple levels for each game definition using the representation defined in the fourth block.

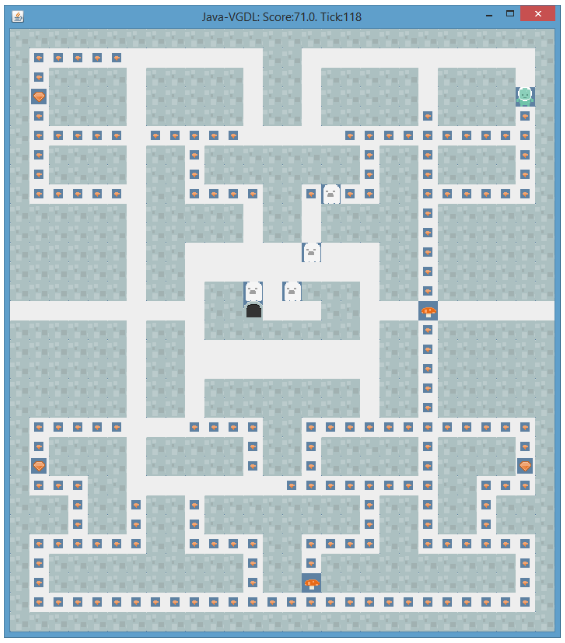
Even though this is a rather simple description of a game, many different games can be created using this description. Figure 2.2 shows three example games, *Aliens*, an adaptation of Space Invaders, *Zelda* and *Pac-Man* that were used for the GVG-AI Competition in 2014.



(a) Aliens



(b) Zelda



(c) Pac-Man

Figure 2.2: Visualization of games for the GVG-AI Competition

These games already indicate the variety of possible games. All three example games have different winning conditions, scoring mechanisms and a different number of NPCs. In *Zelda*, the agent has to get the key and move to the exit without getting killed. Obtaining the key and exiting with the key increases the score by one point each. The agent earns two additional points by killing enemies with a sword. If the player is killed by an enemy, the score is decreased by one point. In *Aliens*, the agent can only move horizontally and shoot at enemies or walls. The agent wins if all enemies are killed. Hitting a wall or an enemy increases the score by one point, getting hit by an enemy ends the game and decreases the score by one point. In *Pac-Man* the agent has to eat all objects, such as pellets, in a maze. The agent is chased by ghosts, that can only be eaten in a short period after eating a power-pill. Collecting different objects increases the score, getting killed by a ghost decreases the score. These games already show complex winning conditions and scoring systems. In *Zelda* and *Pac-Man*, the order of collecting objects is important. Playing *Zelda*, the agent has to get the key before exiting the level. Playing *Pac-Man*, he has to eat power-pills to kill enemy ghosts to increase the score. In both games, the behavior of an object, such as the exit and the ghost, change after collecting another object, such as the key or the power-pill. In *Aliens*, the NPCs are spawning from a portal and can be killed by the agent. Thus the number of NPCs is changing during the course of the game.

To test the agents, the GVG-AI Competition provides different sets of games. At the time of writing, there are three training sets for the competition in 2015. Two of them were used for the competition in 2014, a third set that has no NPCs was created for the competition in 2015. The training sets, including the game and level definition, can be downloaded with the framework. The first two sets of 2014 contain a variety of different games, such as adaptations of popular video games as well as new ones. The training set of 2015 contains only games without NPC, referred to as "puzzles" (Perez, 2015).

In addition to the training set, an agent can be tested against a validation set. This set contains unknown games and is not available for download. The agent has to be uploaded to the homepage for the GVG-AI Competition for testing against the validation set. This prevents the agents to overfit on known games and ensures that an agent is able to execute on the server for the competition.

This subsection mentioned some characteristics of the games for the GVG-AI Competition with resulting challenges. To give some ideas for new agents, the framework contains examples of agents that are using common AI techniques. The next section introduces these example agents.

2.3.3 Sample Agents

The framework provided for the GVG-AI Competition contains different agents as an example of how to create new agents. It includes agents that are helpful to understand the games and replay them, as well as agents that use common AI techniques to perform an action. The agents participated in the competition in 2014 and some of them even got good results.

Two agents, the HUMAN PLAYER and the REPLAYER, can be used to get a better understanding of a game and for debugging. The HUMAN PLAYER allows a human to control the avatar by using the keyboard. This allows to get a better understanding of the game mechanics and is a nice motivation for the developer. The REPLAYER can be used to replay a game based on an action file that can optionally be created while playing a game.

The other agents provide an insight in how to implement agents in the framework and how to apply common AI techniques. The simplest agent is the RANDOM PLAYER. It performs a random valid action without further evaluation. It includes a simple time management to utilize more available time in the `act` method. Another simple approach is the ONE STEP LOOK AHEAD PLAYER. Starting at the current state, the agent simulates all valid actions and selects the most promising one, based on the score provided by the forward model.

The remaining three sample agents use MCTS and a Genetic Algorithm (GA) to obtain an action every game cycle. Two agents, the SAMPLE MCTS PLAYER and the SAMPLE OL-MCTS PLAYER, use a similar MCTS implementation. The SAMPLE MCTS PLAYER is a MCTS based player that is extended with common enhancements, which are explained in detail in Chapter 3. The SAMPLE OL-MCTS PLAYER is similar to the SAMPLE MCTS PLAYER, but uses an "Open Loop" (Perez *et al.*, 2015) implementation. Thus, the agent focuses more on information gained from the current state of the game, by, for example, using the root state as start for the play-out. No information of the state is stored in the nodes during MCTS. All subsequent states are generated using the forward model. Section 5.3 evaluates the advantages of this approach for the GVG-AI Competition. The GA PLAYER uses a microbial GA (Harvey, 2011) that differs from the approach described in Chapter 4. During the GVG-AI Competition in 2014, the SAMPLE MCTS PLAYER was ranked third and performed best of all sample agents.

All agents are a good starting point to create a custom agent as they show examples of how to implement an agent with components such as time management. Some of the used heuristics, like a scoring system that rewards wins and penalizes losses, can be used for other agents. Furthermore, the performance of the sample agents and the competition's results of 2014 can be used for benchmarking and improving an agent.

2.4 Game Characteristics

Games have a variety of characteristics that can have influence on the choice of AI techniques to play them. Important characteristics are, for example, determinism, the number of actions that can be performed or more domain-specific ones like the size of the game's world. This thesis focuses on general game playing, thus games can have a variety of different characteristics. Nevertheless, the context of the GVG-AI Competition provides information about possible game characteristics that have to be considered for the implementation of game playing agents. This information can be obtained by looking at the rules, the framework and underlying components, such as the VGDL.

All existing games of the GVG-AI Competition are fully observable. The forward model of the competition's framework provides all positions of all different game objects, such as NPCs, items or walls. However, the agent has incomplete information about the games. As an example, the agent does not know about rules that define the winning condition or the outcome of a collision of two game objects. Still, the forward model provides information about the score and a list of legal actions. Furthermore, the forward model shows whether a game is over or not and who won the game.

Concerning determinism, the games of the GVG-AI Competition can be both: deterministic as well as non-deterministic. On the one hand, there are deterministic games like *Painter*, where the agent can move across the map and each action is predictable. On the other hand there are non-deterministic games, such as *Aliens*, where the NPCs shoot randomly towards the ground. Other non-deterministic games have aspects like random moving NPCs. Most information about these characteristics can be investigated by using the forward model. The forward model allows to simulate an action based on a game state. This process follows the same deterministic or non-deterministic rules of a game and gives one possible outcome. Thus, executing an action starting at a certain game state can have multiple possible outcomes. The forward model returns only one of them and gives additional information, such as the mentioned score.

Even though the score information is provided by the forward model, it is not normalized and there is no information about the possible range of scores. So there can be games, such as *Frogs* with a score of 1 for a win and 0 for every other game state or games with a broader score range such as *Pac-Man* (Perez *et al.*, 2015). Similar to the range of scores, the number of possible actions in a game can vary. The number of actions is limited to a maximum of seven actions. Especially compared to other domains, where the action space can be continuous, this number of available actions is rather small. Therefore, using these actions to build a game tree results in trees with a rather small branching factor. Nevertheless, sequences of actions can give one action different characteristics. For example, the *ACTION_USE* can change after collecting objects such as a key. Thus, the importance of one action can change over time.

Another characteristic of the games is a result of the game description. The current version of the VGDL allows to build two dimensional maps of different sizes (Perez *et al.*, 2015). Examples of the existing game sets provided for the GVG-AI Competition are *Painter*, a puzzle with small maps beginning at 4×3 squares, or *Pac-Man* with a map up to 28×31 squares. The different map sizes can cause different challenges such as the horizon effect for large maps. Another aspect of the map is that not all positions can be reached by the agent. Walls can build mazes and other objects can obstruct these paths. Some games, such as Boulder Dash where the agent has to dig its way to certain objects, can even have changing paths. Although the length of a game is limited to 2000 game cycles, different characteristics could cause games to end much earlier resulting in a win or loss. These and other characteristics resulting from the game description, such as a unknown number of NPCs, can have influence on the performance of agents.

Chapter 3

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) has been a popular search technique for agents playing games and can be enhanced in several ways. This chapter explains all steps of the search technique in its simple form, as well as different strategies and their enhancements. Thereby, MCTS is also discussed in relation with the GVG-AI Competition.

Chapter contents: Overview, Strategies, Enhancements and Influence of GVG-AI Competition on MCTS

3.1 Overview

MCTS is a best-first search technique that gradually extends the search tree by combining Monte-Carlo evaluation with tree search (Coulom, 2007). The search technique is commonly used for agents playing specific games, such as Go or video games, in which they have been performing well (Browne *et al.*, 2012). The algorithm in a basic form does not require the use of domain knowledge and can be extended in several ways (Powley, Cowling, and Whitehouse, 2014). Thus, it is applicable for general game playing and agents like the CADIA-PLAYER even won competitions such as the GGP Competition (Finnsson and Björnsson, 2008).

MCTS can be divided into four steps: selection, expansion, play-out and backpropagation. These steps are repeated several times for a certain amount of available time to find a promising action. Thus, MCTS is able to return a reasonable action in a short amount of time, as each iteration usually takes a short amount of time. If more time is available, MCTS will find a more promising action based on more iterations of the mentioned steps.

Figure 3.1 shows all steps of MCTS starting after six iterations. The number at the top of the nodes represents the number of wins, the number at the bottom shows the overall number of visits. The selection step starts at the root node of the search tree and tries to select a promising node based on information gathered during past iterations. Promising nodes can, for example, be nodes that obtained a high average score or nodes that have been explored less. Many approaches for selection, such as the Upper Confidence Bounds applied to Trees (UCT) described in Subsection 3.2.1, can be configured to focus more on exploration of less known nodes or exploitation of promising nodes.

After a node is selected, the search tree is expanded by one or more new nodes. Similar to the selection, the expansion step can have different heuristics to choose between possible new nodes. A simple approach is to expand the tree by using one or more unexplored random actions and adding nodes for each action. More complex approaches, such as the ones explained in Subsection 3.2.2, can use statistics gathered during previous iterations to expand promising nodes.

Starting at the added node, the play-out simulates a possible course of the game. In the basic algorithm, random actions are executed until the game is over or until a certain play-out depth is reached. More advanced techniques, like the techniques mentioned in Subsection 3.2.3, guide the play-out to, for example, shorten the play-out and thereby increase the overall performance of MCTS. As an example, these techniques can apply actions based on gathered statistics or try to prevent reoccurring situations.

After the play-out is finished, a certain game state is reached. This state can have different results. The game can be won, lost or the state can be unknown because the play-out had to be interrupted due

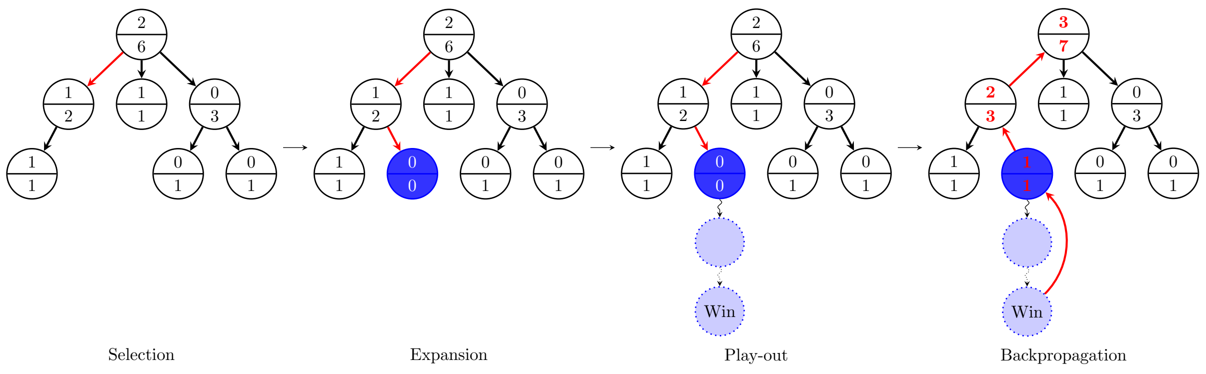


Figure 3.1: Steps of MCTS according to Chaslot *et al.* (2008)

to, for instance, time constraints. Sometimes additional information, such as domain-specific heuristics to score the state of a game can be incorporated to differentiate the results. This information is backed up in the last step of MCTS. Starting at the node that was added during expansion, the results are updated for all parent nodes. Common statistics that are gathered are the frequency of a node being visited, the score – if available – and the number of times the outcome of the play-out was a win. Some statistics that can be gathered are discussed in Subsection 3.2.4.

The root node is created at the beginning of MCTS and no play-out is performed. Thus, the number of visits is equal to the summarized number of visits of the children. All other nodes are evaluated once during play-out. As a result, the number of visits is equal to the summarized number of visits of the children plus one visit for the play-out starting at the node.

Many strategies and enhancements for all four steps of MCTS have been proposed during the last years (Browne *et al.*, 2012). The next section introduces the main ideas behind strategies that are used for the MCTS based agent. Section 3.3 explains different enhancements of these strategies that are evaluated in Chapter 5.

3.2 Strategies

3.2.1 Selection

The major goal of a selection strategy is to find a good balance between exploration and exploitation of game states. A strategy that focuses more on exploration creates wider search trees. A strategy that focuses on exploitation creates deeper search trees. Of special interest are techniques that exploit only promising paths in more depth and explore paths that are of unknown interest to a certain extent. Thereby, a more unbalanced tree is built. Different strategies that have influence on this behavior inside the selection step are explained in this subsection.

Upper Confidence Bounds for Trees

A popular enhancement for selection, which is, for example, used by the SAMPLE MCTS PLAYER of the GVG-AI Competition, is the Upper Confidence bounds for Trees (UCT) (Kocsis and Szepesvári, 2006). The main idea of the algorithm is to use information gathered during previous iterations of MCTS to decide whether to explore a new child or exploit a promising child. The child with the highest UCT value according to Equation (3.1) is selected.

$$UCT_i = \overline{X_i} + C \cdot \sqrt{\frac{\ln(n_p)}{n_i}} \tag{3.1}$$

$\overline{X_i}$ is the average score for child i. A high average score can be decisive for exploiting a child. The variable n_i is the number of times child i has been visited, n_p is the number of times the parent has been visited. The fractions gets smaller, the more often child i has been visited. The term therefore can be interpreted as the value for exploration. C is a constant that can be used to tune the balance between exploration and exploitation. Higher values for C result in a selection that focuses more on exploration, lower values encourage exploitation.

ϵ-Greedy Selection

The ϵ-Greedy (Sutton and Barto, 1998) selection strategy is a combination of a random selection and a greedy selection strategy. The parameter ϵ has a value between 0 and 1 that represents the probability of selecting the next state randomly. This random component uniformly selects states, amongst them unknown states, and thus ϵ controls the exploration. With the probability of $1 - \epsilon$, a child with the highest average score is chosen and thus exploited.

Random Selection

The simplest strategy to select a child is the random selection. Albeit the simplicity, this technique can be implemented in different ways. One approach is to choose a random action at each node, starting at the root node. If the corresponding child for the action is inside the tree, the selection process continues with this child. Otherwise, the process stops and returns the last selected node inside the tree.

Another approach, which is used in this thesis, is to investigate all children of a node once before investigating its grandchildren. A node is selected if not all possible actions have been investigated. Otherwise, the strategy continues with a random child. For the random selection, no statistics have to be kept in memory and no additional calculations are necessary. Thus, the choice for one action during one iteration of the random selection has less computational effort compared to all other strategies. However, the overall performance of MCTS can be worse compared to other strategies as the selection is not optimal. A guiding selection strategy that tries to reach a terminal game state can reduce the number of performed actions and therefore increase the overall performance of MCTS.

3.2.2 Expansion

After the selection strategy returns a node, there are different ways to expand this node. Similar to the selection, one or more children can be added to the search tree randomly or based on statistics. The major difference to the selection step is that no information from previous iterations of MCTS is available for a specific child. The simplest way to expand the search tree is to add only one child. This is done by many MCTS based agents after the selection and before the play-out. Another possibility is to expand after the play-out (Chaslot *et al.*, 2008). This allows to add the complete path traversed during the play-out to the search tree.

The major task for an expansion strategy is to choose the children that should be added to the search tree. The MCTS based agent for this thesis expands after the selection and before the play-out by using different heuristics. The remainder of this section summarizes the developed strategies for the MCTS based agent that decide which nodes are expanded and thus kept in memory.

The RANDOM EXPANSION randomly chooses one of the unexplored actions. The node that is obtained by executing the action starting at the selected node is added to the tree. It is the simplest strategy and it is used by MCTS based agents, such as the SAMPLE MCTS PLAYER and the SAMPLE OL-MCTS PLAYER.

With the probability of ϵ, the ϵ-GREEDY EXPANSION chooses a random unexplored action. The corresponding node for executing the action is added to the tree. With the probability of $1 - \epsilon$, all unexplored actions are assessed by using the forward model. The resulting state with the highest score obtained by the forward model is added to the tree. This strategy could be modified to look only at a certain number of actions to reduce the number of state evaluations.

Similar to the ϵ-GREEDY EXPANSION, the ACTION-BASED EXPANSION adds children to the search tree based on their MAST values. This strategy uses the average score that was obtained by applying an action during a play-out for choosing a promising action. With the probability of $1 - \epsilon$, the tree is expanded by applying the unexplored action with the highest average score. Otherwise, a random action is chosen.

All previous strategies add one node at a time to the tree. Using the ALL CHILDREN EXPANSION strategy, all actions provided by the forward model are used to expand the search tree. This is possible in the context of the GVG-AI Competition as the number of possible actions is limited to a maximum of seven. Thus, the time necessary for expansion is less than in other domains where the number of actions is much higher or even continuous. One of the added nodes is chosen randomly for starting the play-out.

3.2.3 Play-out

The play-out strategy tries to guide simulations starting at the node provided by the expansion strategy. A play-out may have three categories for the outcome. Either, the game is won, lost or it is unknown due to restrictions in the simulation depth. In case of a prematurely stopped simulation, available information or heuristics can be incorporated to obtain a score for a game state. Nevertheless, a win or a loss is a more meaningful result, as many game states can have similar scores. It is even possible that game states with high scores have a high probability to be lost during the next steps.

The performance of a play-out strategy is an important aspect. On the one hand, a fast strategy allows to perform more play-outs than a slow one. Thus, fast and simple play-out strategies could perform better than complex and computation intensive strategies. On the other hand, a more complex strategy can guide the play-out to a more reasonable simulated course of the game. Thus, the obtained score can be more accurate. The play-out can become even shorter and therefore faster, as for example possible win situations are recognized. This section presents both, simple and more informed play-out strategies that are implemented for the MCTS based agent.

Random Play-out

The random play-out strategy is the standard strategy used in MCTS. Starting at the game state that was added in the expansion step, a random valid action is chosen. Random actions are applied until the game is over or a maximum play-out depth is reached. This maximum depth is a threshold to prevent the play-out from spending too much time. The available actions for the agent and NPCs to choose from are usually known or can be deduced by given rules of a game. Similar to the random selection described in Subsection 3.2.1, the play-out strategy is easy to implement. Furthermore, one step of the random play-out needs least computational resources. Again, the overall performance can be worse, as a guiding strategy during the play-out can shorten the process and thus increase the overall performance.

MAST

The Move-Average Sampling Technique (MAST) by Finnsson and Björnsson (2008) uses action-based statistics to guide the play-out. The strategy is a promising approach that was successfully used by the CADIA-PLAYER of Finnsson (2007), an agent that won the GGP Competition in 2008. Instead of applying random actions during the play-out, promising actions can be selected according to average scores gathered in previous simulations. The original version of MAST uses a Gibbs distribution to choose more likely actions with higher average scores. Instead of using the Gibbs distribution, the implementation for this thesis chooses a random action with the probability of ϵ (Tak, Winands, and Björnsson, 2012). Otherwise, the action is chosen greedily. Thus, ϵ can be used to tune the balance between exploration and exploitation. Gathering the additional action-based statistics has low impact on performance, especially as the number of actions is limited in the GVG-AI Competition.

3.2.4 Backpropagation

The main purpose of the backpropagation is to store all necessary statistics that can be used by all other steps. Thus, the backpropagation is dependent on these steps and can have different calculations for gathering these statistics. Most strategies, for example, need a certain score of a node and the information how often the node has been visited. Equation (3.2) shows a basic approach by Chaslot *et al.* (2008).

$$\text{score} = \begin{cases} 1 & \text{, if agent wins} \\ 0 & \text{, draw} \\ -1 & \text{, if agent loses} \end{cases} \tag{3.2}$$

The value of a node is increased by 1 if the agent wins, decreased by 1 if the agent loses and stays the same in case of a draw. Dividing the cumulative score by the number of times the node has been visited, can be used as an indication of its quality. Although this heuristic can be used by most games, there is no information about the score of a prematurely interrupted play-out. Nevertheless, most enhancements described in Subsection 3.3.3 use a similar idea.

3.3 Enhancements

3.3.1 Selection

Upper Confidence Bounds for Trees

The basic UCT formula allows to tune the parameter C to balance between exploration and exploitation. For a specific game, it is possible to determine a value for C that performs well as characteristics such as the range of possible scores are fixed. In general game playing, the scores of a game can be more diverse, thus the influence of C in relation to the average score changes. Different adaptations of UCT are available that use other approaches to change this influence. Equations (3.3) to (3.6) show some examples that have been implemented for the MCTS based agent.

$$UCT_i^{\text{Adaptive C}} = \overline{X}_i + C \cdot (\max(X_i) - \min(X_i)) \cdot \sqrt{\frac{\ln(n_p)}{n_i}} \tag{3.3}$$

Equation (3.3) by Schepers (2012) simply scales the value for C by the range of scores for a child. The higher the range of values that have been seen so far, the more variation is possible in the average score of the child. Thus, the exploration of this child should be favored. The maximum and minimum scores can be obtained by looking at global scores or scores of one child.

$$UCT_i^{\text{UCB1-Tuned}} = \overline{X}_i + \sqrt{\frac{\ln(n_p)}{n_i} \cdot \min\left\{\frac{1}{4}, \left(\frac{1}{n_i}\sum_i X_i^2\right) - \overline{X}_i^2 + \sqrt{\frac{2\ln(n_p)}{n_i}}\right\}} \tag{3.4}$$

A similar approach named UCB1-TUNED is described in Equation (3.4) by Auer, Cesa-Bianchi, and Fischer (2002). UCB1-TUNED replaces C of the basic UCT formula by an additional calculation to change the behavior between exploration and exploitation. The equation changes the exploring part towards nodes where the variance in scores – $\left(\frac{1}{n_i}\sum_i X_i^2\right) - \overline{X}_i^2 + \sqrt{\frac{2\ln(n_p)}{n_i}}$ – is larger. The original paper proposes an empirically evaluated upper bound of $\frac{1}{4}$ in this part.

$$UCT_i^{\text{Progressive History}} = \overline{X}_i + C \cdot \sqrt{\frac{\ln(n_p)}{n_i}} + \frac{X_a}{n_a} \cdot \frac{W}{n_i - X_i + 1} \tag{3.5}$$

Another idea by Nijssen and Winands (2011) is called Progressive History. The formula described in Equation (3.5) adds an action-based score that decreases in influence after more games are played. The term $\frac{X_a}{n_a}$ is the average score obtained in every simulation where action a has been played. The fraction $\frac{W}{n_i - X_i + 1}$ decreases over time and the overall influence can be tuned with W. Higher values for W bias the overall score towards the action-based score. The term $n_i - X_i$ reduces the influence of actions that do not perform well over many iterations. Thus, the scores for this formula have to be between 0 for a loss and 1 for a win to obtain this behavior.

$$UCT_i^{\text{Single-Player}} = \overline{X}_i + C \cdot \sqrt{\frac{\ln(n_p)}{n_i}} + \sqrt{\frac{\sum_i X_i^2 - n_i \cdot \overline{X}_i + D}{n_i}} \tag{3.6}$$

Equation (3.6) by Schadd *et al.* (2008) adds another term to the original UCT formula. This term consists of the squared results $\sum x^2$ of the child subtracted by the expected results $n \cdot \overline{X}_i$ and an additional constant D. The term is added to represent a deviation of scores for the child node. The constant D is added to explore less visited nodes. Most applications of MCTS are games that have scores of -1 for a loss, 0 for a draw and $+1$ for a win. The games of the GVG-AI Competition can have a different scale of scores which makes this formula interesting as it is able to compensate the variations in scores.

ϵ-Greedy Selection

The basic ϵ-greedy selection uses a constant value for ϵ, thus has a fixed balance between exploration and exploitation. Auer *et al.* (2002) proposes to use $\epsilon = \frac{1}{n}$ where n is the iteration of MCTS for the current move. Thus, the probability to choose random actions decreases each iteration and the exploitation is encouraged.

Evo Selection

Another approach to enhance MCTS, which was suggested by, for example, Benbassat and Sipper (2014) and Perez *et al.* (2014), is to use an evolutionary algorithm to get additional information about good actions. The main idea is to learn additional features of a game during the play-out of MCTS. These features can then help to decide which actions lead to promising states. The obtained model can be used to guide the play-out and the selection step of MCTS. The detailed description of the approach chosen for this thesis is presented in Chapter 4. Amongst the mentioned techniques for selection, this approach is the one that needs the most computational resources.

3.3.2 Play-out

N-Gram Selection Technique

Similar to MAST described in Subsection 3.2.3, the N-Gram Selection Technique (NST) by Tak *et al.* (2012) looks at the last N applied actions and stores statistics for them. For $N = 1$, the N-gram approach is similar to MAST. During the backpropagation, NST stores scores for each occurred sequence of actions of length N up to the root. The original implementation stores scores for one action, 2-grams and 3-grams of actions. Based on the average of these scores, the next action is chosen during play-out.

Starting at the added node after expansion, s scores are looked up in the statistics for action sequences with length $l \in [1, s]$. Analyzing the last $l - 1$ parents of the added node gives information about the last $l - 1$ actions. Using the available legal actions at the current state obtained by the forward model, the l^{th} action of the N-gram is chosen. The average of the scores for all s N-grams determines the value for the l^{th} action. The scores of lengths other than 1 are only taken into account if they have been visited at least k times. Tak *et al.* (2012) use an empirically determined threshold of $k = 7$. With the probability of $1 - \epsilon$, the action with the highest average score is chosen. Otherwise, a random action is chosen. To bias the strategy towards unexplored actions, a high default score is set.

The action statistics can be gathered over multiple simulations. Thus, action scores of later game have less influence on the average score. As the importance of an action can change over time, a decaying strategy can be interesting. Tak, Winands, and Björnsson (2014) propose three different methods to decay the statistics. The first method, called MOVE DECAY, uses a decay factor $\gamma \in [0, 1]$. The value for an action is multiplied by this factor after an action is chosen. The second method, called BATCH DECAY, makes use of the same update strategy for scores, but is only executed after a fixed number of simulations. The SIMULATION DECAY decays only statistics of N-grams that were played in the simulation. N-grams that occurred multiple times are decayed multiple times after each simulation. According to Tak *et al.* (2014), the MOVE DECAY is suspected to perform best. Thus, MOVE DECAY is used in this thesis as decaying strategy.

Evo Play-out

Similar to MAST that uses information about actions during the play-out, additional information provided by a genetic algorithm can be used. The main idea is to learn additional properties of a game state that can be used to determine which next state is promising during the play-out. These properties, such as the number of NPCs or the current position of the avatar, can be used to learn rules for game states that can guide the play-out. This approach is the most complex one due to additional calculations inside the genetic algorithm. The approach as well as additional background information and enhancements that tackle this challenge are explained in detail in Chapter 4.

3.3.3 Backpropagation

The statistics that are updated during backpropagation can be used by other steps of MCTS. Some strategies are even expecting a certain range for scores. Thus, the calculation of the score and other statistics have big influence on MCTS.

The score and the information whether a game is won or lost is provided by the forward model in the GVG-AI Competition. Combining this information gives a more granular score than the basic score based on the information whether a game is won or lost. Equation (3.7) shows a heuristic for scoring a game state that is used by the sample players of the GVG-AI Competition, such as the SAMPLE MCTS PLAYER. The heuristic uses the score and the winner provided by the forward model and combines them to one value. According to the rules of the competition that are described in Section 2.2, the score is increased when the agent wins and decreased when it loses. To use this score in, for instance, the UCT selection, it has to be normalized to a range from 0 to 1 by scaling it to the maximum and minimum score obtained so far.

$$\text{GVG-AI Score Heuristic} = \begin{cases} \mathit{score} + 10000000 & \text{, if agent wins} \\ \mathit{score} - 10000000 & \text{, if agent loses} \\ \mathit{score} & \text{, otherwise} \end{cases} \tag{3.7}$$

Most strategies, such as UCT, require information about how often a node is visited and the accumulated score that was obtained during the play-out. Other strategies, such as strategies based on the idea of MAST, need additional statistics like the average score obtained by applying an action. Therefore, the backpropagation is selected according to the minimal information necessary for all other steps. Furthermore, some strategies need normalized scores. For example, the basic UCT Selection is designed for scores between 0 and 1. Thus, later versions of the SAMPLE MCTS PLAYER normalize the score from Section 2.2 to observed minima and maxima of scores before using it for the calculation of UCT values. Some agents extend this range from −1 to 1 (Chaslot *et al.*, 2008). Equation (3.8) shows a normalized version of the heuristic used by the GVG-AI Competition.

$$\text{Normalized Score Heuristic} = \begin{cases} 1 & \text{, if agent wins} \\ -1 & \text{, if agent loses} \\ -1 + \frac{2}{1+e^{-k\cdot \mathit{score}}} & \text{, otherwise} \end{cases} \tag{3.8}$$

When the game is won, the maximum score of 1 is returned. When the game is lost, the minimum score of −1 is returned. Otherwise, the score is normalized by using a sigmoid function (Pepels *et al.*, 2014a). This function normalizes the score to a range from −1 to 1 centered around 0. The parameter k can be used to tune the influence of the score towards the maximum and minimum score. Values for k closer to 0, for example, push the value of the heuristic towards 0.

Pepels *et al.* (2014a) use the sigmoid function to normalize two additional values, a relative bonus and a qualitative bonus. The relative bonus b_r uses the depth as quality measure, the qualitative bonus b_q uses a quality assessment q of a play-out's result. These bonuses can be incorporated to a score heuristic as described in Equation (3.9).

$$\text{Quality-Based Score Heuristic} = \begin{cases} 1 + b_r + b_q & \text{, if agent wins} \\ -1 + b_r + b_q & \text{, if agent loses} \\ 0 + b_q & \text{, otherwise} \end{cases} \tag{3.9}$$

$$\text{Relative Bonus } b_r = \text{sgn}(s_{\{-1,0,1\}}) \cdot a \cdot \text{sigmoid}(\lambda_r) \tag{3.10}$$

$$\text{Qualitative Bonus } b_q = \text{sgn}(s_{\{-1,0,1\}}) \cdot a \cdot \text{sigmoid}(\lambda_q) \tag{3.11}$$

$$\text{sigmoid}(x) = -1 + \frac{2}{1+e^{-k\cdot x}} \tag{3.12}$$

$$\lambda_r = \frac{\overline{D} - d}{\sigma_D} \tag{3.13}$$

$$\lambda_q = \frac{q - \overline{Q}}{\sigma_Q} \tag{3.14}$$

The bonuses b_r and b_q in Equations (3.10) and (3.11) are used to incorporate additional information into the overall reward. The variable $s_{\{-1,0,1\}}$ is -1 when a game is lost, 1 when a game is won and 0 in case of a draw. Pepels *et al.* (2014a) propose a method to compute a or choose an empirically determined value, for example 0.25. Equations (3.13) and (3.14) normalize the value used for the bonus by using the average and the standard deviation σ of the value. The variable d is the depth of the last node of a simulation, thus a combination of the depth in the search tree and the play-out depth. Accordingly, $\overline{D}$ is the average combined depth of all play-outs. If the depth d is above the average $\overline{D}$, the normalized value λ_r gets negative. Thus, the result of the sigmoid function is negative. In case of a win, the sign function returns a positive value and the total relative bonus is negative. In case of a loss, the sign function returns a negative value and the total relative bonus is positive. For values of d below the average $\overline{D}$, the behavior is the opposite. Therefore, the relative bonus favors short wins and long losses and gives penalties for long wins and short losses.

The average and the standard deviation can be collected for a single simulation of MCTS or over the whole game. Pepels *et al.* (2014a) propose to reset the statistics after selecting an action. The variable $q \in [0,1]$ can be domain-specific knowledge, such as the score provided by the forward model in the GVG-AI Competition or other heuristics. The average of these quality measures is represented by $\overline{Q}$. For performance reasons, σ can be estimated by

$$\sigma^2 \approx \hat{\sigma}^2 = \frac{\sum_i x_i^2 - \frac{\left(\sum_i x_i\right)^2}{n}}{n-1}$$

This unbiased estimate of the variance can be updated iteratively with values for x_i. The variable x_i in this case are gathered values for d or q. The variances and averages take some time to get reasonable values. Thus, the bonuses are only incorporated if the variance is below a certain empirically determined threshold δ.

The quality-based heuristic was originally designed for two-player games. Differences in the design of, for example, the structure of search trees can influence the behavior of the heuristic. This thesis therefore evaluates if a separation of statistics for the two bonuses is helpful for the GVG-AI Competition. The idea is to keep separate statistics for bonuses depending on the state of the game. Thereby, a won state is evaluated with statistics obtained by other won games so far. Otherwise, other statistics are used for draws and losses. The results of the adaptation are presented in Subsection 5.2.3.

3.4 Influence of the GVG-AI Competition on MCTS

3.4.1 Influence on Strategies

Many characteristics of the GVG-AI Competition, such as the rules, the framework and the resulting game characteristics described in Section 2.4, have influence on the MCTS based agent. The competition, for example, does not allow multithreading that is used by some enhancements. Thus, enhancements that are against the rules of the competition are not discussed. Furthermore, the characteristics of the games, such as an unknown number of NPCs that can perform several actions at one game step have influence on the agent's performance. This section presents these aspects of the competition as well as their influence on MCTS.

Most strategies and enhancements, such as basic UCT selection, are developed for a specific game. Thus, it is possible to tune parameters, such as C, for the specific domain. As the games for the GVG-AI Competition are not known, C cannot be optimized for the specific games. Either, a value has to be chosen in advance based on the test and training set of games, or the parameters have to be learned online. Another approach, which is used by some enhancements, is to replace these parameters by other formulas or scale them according to game statistics. For example, the Evo play-out tries to learn rules for state evaluation to guide the play-out. The main challenge is to find a good approach with adequate parameters so that the algorithm is still able to compute in the time frame of the GVG-AI Competition.

The GVG-AI Competition limits the time for every move to 40 milliseconds. In other domains and competitions, this period can be much longer. Especially when the game is known a priori, the agent is able to learn certain strategies for that specific game in advance and store them for later use. This has impact on the possible complexity of enhancements in the GVG-AI Competition. As an example, the SAMPLE MCTS PLAYER limits the simulation depth in the play-out of MCTS to ten. For better comparability, the agent for this thesis uses the same depth. Some popular techniques, such as static evaluation functions or evolutionary algorithms, can take much time for computation. This could result in fewer play-outs of MCTS and can therefore decrease the overall performance compared to simpler techniques. Thus, it is important to get a reasonable result in this limited amount of time, even if the algorithm has to be interrupted prematurely. The iterative character of MCTS helps to achieve this goal as each iteration takes a small amount of time to finish.

Another important part of the competition's framework for implementing an MCTS based agent, is the forward model. The forward model provides a score that can be used, for example, inside the steps of MCTS to diversify the quality of game states. In addition to the information whether a state is won or lost, the score can give an indication of how good a state is. One additional challenge is the non-deterministic character of games, like random moving opponents or dice rolls. Starting at the same game state, using the forward model to simulate an action for the first time can have a different outcome than performing it a second time. Thus, it makes a difference whether to store a game state inside the nodes or using the forward model to generate them every time a state is visited. This is of particular importance in the selection step, as applying one action at the same state can result in different consecutive states. Techniques that do not store the information of the state inside the nodes are often referred as "Open Loop", other techniques that store the game state inside the node are called "Closed Loop" (Perez *et al.*, 2015). The presented selection strategies can be implemented in both ways. The winner and the third ranked agent of the GVG-AI competition in 2014 use an "Open Loop" approach. Thus, the approach is chosen for this thesis and it is further evaluated in Section 5.3.

Furthermore, the forward model provides a list of available legal actions and is able to simulate the moves of all NPCs. Other competitions provide this information by showing the rules of a game, for example, in GDL. Therefore, the MCTS based agent for the GVG-AI Competition does not need to learn legal moves for any player. In addition, the number of possible actions in the GVG-AI Competition is limited to seven. Compared to other domains, where the number of actions can be higher or even continuous, this number of actions is small. On the one hand, this simplifies some techniques in so far as action discretization is not necessary. It is even possible to expand all actions in a short amount of time. On the other hand, the characteristics of the actions can change over time. The *ACTION_USE*, for instance, can change after collecting an item, such as a key. This could influence strategies such as MAST that keep action-based statistics.

In addition to the winner of a game, the forward model provides a score information. Although this additional score helps to estimate the quality of a game state, additional techniques have to be used to normalize this score. Due to the variety of games and their scores, normalization can be necessary for some strategies. The combination and evaluation that are presented in the experiments of Chapter 5 indicate which approach is applicable for general game playing.

3.4.2 Architecture of MCTS based Agent

As the GVG-AI Competition was held first in 2014, there is less information and evaluation of techniques available compared to older competitions, such as the GGP Competition. The results of agents based on the existing game sets give a good indication about the performance of each agent. However, the results do not show the effect of enhancements for each agent. Thus, this thesis evaluates different enhancements for MCTS. This section describes the overall structure of the agent that allows to combine and compare different enhancements. Furthermore, some example algorithms are explained in the context of the GVG-AI Competition.

Algorithm 3.1 shows the four basic steps of MCTS in the developed agent. In addition, a simple time management is shown, that was taken from the SAMPLE MCTS PLAYER for better comparability. The agent basically keeps track of the average time taken for each iteration and checks whether the remaining time is enough to perform two more iterations. To prevent the agent from being disqualified, an additional `TIME_BUFFER_MILLISEC` of five milliseconds is used in the time management. After the last iteration, the most promising action, based on, for example, the average score of a node, is returned.

Algorithm 3.1 Steps of MCTS

```
Require: $elapsedTimer$: timer that gives information about the remaining time to choose an action
Require: $root$: root node to start MCTS
1: repeat
2:     $iterationTimer \leftarrow$ new ElapsedCpuTimer()
3:     $nodeToExpand \leftarrow selectionStrategy.$getNodeToExpand($root$, $elapsedTimer$)
4:     $addedNode \leftarrow expansionStrategy.$expandNode($nodeToExpand$)
5:     $playOutResult \leftarrow playoutStrategy.$runPlayout($addedNode$, $elapsedTimer$)
6:     $backpropagationStrategy.$backup($expandedNode$, $playOutResult$)
7:     $iterations \leftarrow iterations + 1$
8:     $timeTaken \leftarrow timeTaken + iterationTimer.$elapsedMillis()
9:     $avgTimeTaken \leftarrow timeTaken/iterations$
10:    $remaining \leftarrow elapsedTimer.$remainingTimeMillis()
11: until $remaining > 2 \cdot avgTimeTaken$ && $remaining > TIME\_BUFFER\_MILLISEC$
12: $bestChild \leftarrow$ child node from $root$ with highest average score
13: return $bestChild.$get_actionFromParent()
```

Each strategy for a step of MCTS is represented by an interface. Thus, the strategies can be combined in different ways and compared easily as the overall structure of the algorithm is the same. The different strategies can then be combined to a `MctsController`, that performs the steps as mentioned in Algorithm 3.1. Thereby, MCTS can be reused by other classes, for example to learn different characteristics during initialization. An evolutionary approach, for instance, can use the `MctsController` before playing a game to evaluate individuals based on their performance in MCTS. The relations between the interfaces and other classes of the framework can be seen in Appendix A.

Many different combinations of strategies are possible, though not all are valid. MAST, which can be used in play-out, requires the backpropagation of statistics for actions. The selection strategy returns a node that is passed to the expansion strategy. One or more nodes are then added to the tree and the expansion strategy returns one added node to start the play-out. Based on the last state of the play-out, the statistics are updated. After the time limit is reached, the node with the highest average score is chosen and the associated action is performed.

Algorithm 3.2 shows the overall UCT selection implemented for the agent as an example implementation for the selection strategy. The method `getNodeToExpand` is inherited from an interface that is used by all selection strategies to be exchangeable for testing. The method selects the next child while the game is not over and the node is fully expanded. Children are selected based on the highest UCT value that is calculated in `getUctValue`. The return value is the selected node of the search tree.

Algorithm 3.2 Upper Confidence bounds for Trees for Selection

```
1:  function getNodeToExpand($rootNode$)
2:      $node \leftarrow rootNode$
3:      while !$state$.isGameOver() && $node.unexpandedActions$.size()== 0 do
4:          $node \leftarrow$ getNextNodeByUct($node, timer$)
5:      end while
6:      return $node$
7:  end function

8:  function getNextNodeByUct($node$)
9:      $bestNode \leftarrow null$
10:     $bestUctVal \leftarrow -\infty$
11:     for all $child$ : $node$.get_children() do
12:         $uctVal \leftarrow$ getUctValue($child, timer$)
13:         if $uctVal > bestUctVal$ then
14:             $bestUctVal \leftarrow uctVal$
15:             $bestNode \leftarrow child$
16:         end if
17:     end for
18:     return $bestNode$
19: end function
```

Different UCT calculations, such as the ones mentioned in Subsection 3.3.1, can be achieved by changing the calculation of the value inside `getUctValue`. Similarly, all other strategies are implemented.

In addition to the MCTS specific methods, all interfaces for strategies have a method for initialization. This method is called before the game starts, when the constructor of the agent is called. The framework of the GVG-AI Competition provides a timer and the starting position of the game. With one second, the time-period for initialization is much higher than the 40 milliseconds to choose an action. Strategies can use this time to initialize default scores or to start learning a model based on the starting position of a game.

Chapter 4

Evolutionary Algorithms

Inspired by biology, Evolutionary Algorithms are a family of machine-learning techniques that have been successfully applied to different domains. This chapter describes the principle of operation of Evolutionary Algorithms and different approaches and representations. The techniques are then combined with MCTS for participating in the GVG-AI Competition.

Chapter contents: Overview & Goals, Genetic Programming, Setup in the GVG-AI Competition and Embedding Evolution in MCTS Play-out

4.1 Overview & Goals

Evolutionary Algorithms are techniques inspired by concepts of biology to approximate or solve optimization problems (Koza, 1992). They have been used successfully to a variety of applications, such as mathematical simulations (Li *et al.*, 1996), robotics (Lazarus and Hu, 2001), economics (Arifovic, 1994) and data mining (Freitas, 2003). Evolutionary Algorithms can, for example, be used to evolve rule sets for classification in data mining (Corcoran and Sen, 1994) or to optimize a supply chain (Falcone, Lopes, and dos Santos Coelho, 2008). The variety of applications leads to the idea that the concept could also be useful for general game playing.

Similar to evolution in biology, Evolutionary Algorithms try to approximate a problem and improve their results over generations. Each approximation model is called an individual. The individuals have different characteristics to solve the problem. Many individuals form a population. Inside this population, the individuals have the ability to reproduce or change. The process of combining two individuals is called crossover. The process of changing a single individual is called mutation. In other words, individuals can be combined to new ones or one individual can change characteristics and thus become a new one. The fittest individuals of a generation survive to give a gradually better approximation for the problem.

There are many different types of Evolutionary Algorithms available. This chapter focuses on so-called Genetic Programming, a special category of Evolutionary Algorithms. Two different algorithms of this category are introduced in Section 4.2. Subsection 4.2.1 uses linear functions and Subsection 4.2.2 a tree-based representation for individuals. Each individual tries to represent an evaluation function for a state of a game. The individuals can then be used to learn additional information about the quality of a game state. The goal is to incorporate this information to guide the play-outs of MCTS. The main idea is to shorten the play-outs of MCTS by guiding it to shorter and more reasonable games. Thus, the amount of time for a play-out is less and the overall performance could be increased.

All presented approaches use the assumption that games have certain atomic features. These features can represent different simple aspects of a game, such as the number of objects in the game, the amount of available resources or distances to certain objects. The features can be chosen to be applicable for a variety of games and are thus less domain-specific. Combining these features can give additional information about the game. The tree-based approach combines them by building a tree, the linear function adds values for each feature and weights them.

Perez *et al.* (2014), Benbassat and Sipper (2014) already used Evolutionary Algorithms in combination with MCTS for game playing. Both are using slightly different approaches and representations. Section 4.3 describes these approaches and connects them to the context of the GVG-AI Competition. The introduction of Genetic Programming to the MCTS based agent is then described in Section 4.4.

4.2 Genetic Programming

4.2.1 (1 + 1) Evolutionary Algorithm

Droste, Jansen, and Wegener (2002) have proposed the (1 + 1) Evolutionary Algorithm. The (1 + 1) EA limits the population size to two individuals with a survival rate of 50%. Thus, offspring can only be generated via mutation of the single surviving individual and no crossover is possible. The mutated individual is compared to the original one, hence the name "(1 + 1)", and the fittest individual is the next generation

The variables that can be used for optimization are limited to Boolean values. Thus, each individual $\vec{x}$ is represented by a sequence of n Boolean variables $x_i \in \{0, 1\}$. After the first population has been initialized randomly, next generations can be created by using bit-wise mutation. During the mutation, each bit is flipped independently with a certain probability $p = \frac{1}{n}$. This way, a new individual can be generated that differs in some variables. The goal is to find the individual that maximizes the fitness function $f : \{0, 1\}^n \to \mathbb{R}$

If the fitness of the new individual is higher, the newly generated individual will replace the old one. Otherwise, the old individual is kept. This fitness function can be represented by

$$\begin{aligned} f(x_1, \dots, x_n) &= \sum_{I \subseteq \{1,\dots,n\}} c_f(I) \prod_{i \in I} x_i \\ &= c_f(\emptyset) \\ &\quad + c_f(\{x_1\}) \cdot x_1 + c_f(\{x_2\}) \cdot x_2 + \dots + c_f(\{x_n\}) \cdot x_n \\ &\quad + \dots \\ &\quad + c_f(\{x_1, \dots, x_n\}) \cdot \prod_{i=1}^{n} x_i, \end{aligned}$$

where $c_f \in \mathbb{R}$ is a coefficient that describes the importance of a subset of variables. In other words, the non-trivial goal is to find the subset of Boolean variables that maximizes the fitness function. Droste *et al.* (2002) show that the (1 + 1) EA is an efficient way by using linear fitness functions of the type $f(\vec{x}) = \sum_i^n w_i x_i \to \mathbb{R}$. For linear fitness functions, they have proved that the algorithm is able to solve a problem in log-linear time, depending on the number of Boolean variables.

For the analysis of a game board, for example, it can be interesting how many tokens are on the board. Even more complex features, such as distances between game objects like the avatar and NPCs, can be interesting in general game playing. These features can be interpreted as the coefficients c_f. Using only Boolean variables allows to determine whether a feature is relevant or not. In particular, if all features are positive, maximizing the fitness function is trivial. Thus, existing agents that use different types of Evolutionary Algorithms often use non-Boolean variables to be able to weight features (Benbassat and Sipper, 2010; Perez *et al.*, 2014). To incorporate these weights, the original approach by Droste *et al.* (2002) can be adapted in different ways. The necessary changes as well as additional challenges of general game playing are addressed in Subsection 4.3.1 with an example implementation by Perez *et al.* (2014).

4.2.2 Strongly Typed Genetic Programming

Many Evolutionary Algorithms, such as the (1 + 1) EA, can only use a specific type for their features. Montana (1995) introduces so-called Strongly Typed Genetic Programming (STGP), where all variable and constants get a type in addition to their values. To combine these different features, the representation of an individual is changed. Each individual can be represented as a tree consisting of different nodes. The leaf nodes are called terminal nodes and all intermediate nodes are called non-terminal nodes. Each terminal node returns a value with a fixed type. Each non-terminal node has one or more input types and one return type. A tree can be generated by connecting the nodes depending on input type of the non-terminal nodes and output type of the terminal nodes. A node for integer addition, for example, returns an integer and has two children that both return integers. Similarly many nodes, such as if-then-else nodes or nodes for matrix calculations, can be defined. The root node of the generated trees returns a type that is required to solve a problem. All other nodes return a value with the type required by the parent.

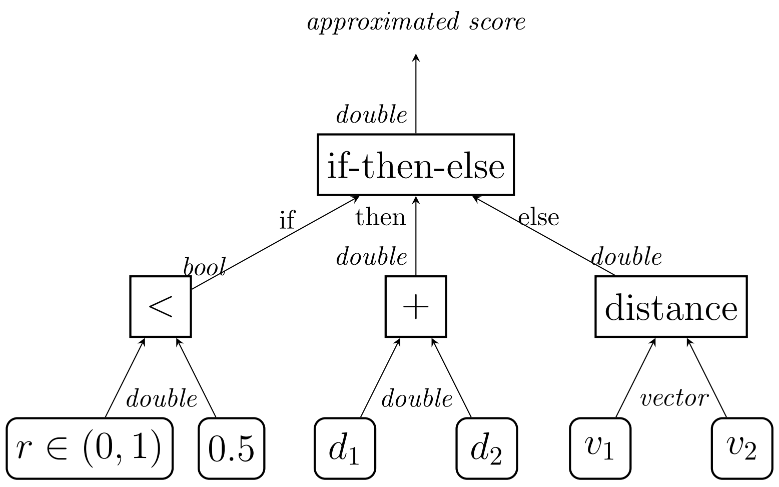


Figure 4.1: Example tree representation of individual in STGP

Figure 4.1 shows a simple example representation of one individual. The tree can be evaluated recursively by starting at the root node. The root node returns a Double value as an approximation for the solution of a given problem. In the example, the root node requires three children. The first child has to return a Boolean value, the other children have to return a numeric type, such as Double. The root node's first child compares two numeric values, one random value in range between 0 and 1 and a constant value of 0.5. If the value for r is smaller than 0.5, the evaluation of the tree will return the sum of two Doubles d_1 and d_2. Otherwise, the distance between two vectors v_1 and v_2 is returned. The values for d and v returned by the terminal nodes can be different features, such as an enemy count for d or positions of game objects for v. Montana (1995) shows different methods that incorporate the type restrictions to grow these trees for initial populations. Furthermore, they explain how to overcome challenges, such as defining generic functions to reuse nodes. The "+" node in the example could for example be used for different types, such as vectors or integers.

Algorithm 4.1 shows a recursive approach to grow trees proposed by Koza (1992). The "full" method creates trees that have all leaf nodes at the maximum depth of the tree. Thus, all nodes that are not at the maximum tree depth have to be non-terminal nodes. The "grow" method generates more unbalanced trees, where leaf nodes can be at any depth up to the maximum depth. Both methods can be used to generate the initial population.

Algorithm 4.1 Generation of trees in STGP

```
function GENERATETREE($maxTreeDepth$, $generation\_method$)
    if $maxTreeDepth == 1$ then
        $root \leftarrow$ random terminal with return type according to problem
    else if $generation\_method == full$ then
        $root \leftarrow$ random non-terminal with return type according to problem
    else
        $root \leftarrow$ random terminal or non-terminal with return type according to problem
    end if
    GENERATESUBTREE($maxTreeDepth - 1$, $root$, $generation\_method$)
    return $root$
end function

function GENERATESUBTREE($remainingTreeDepth$, $parent$, $generation\_method$)
    for all $children$ of $parent$ do
        if $remainingTreeDepth == 1$ then
            $child \leftarrow$ random terminal
        else if $generation\_method == full$ then
            $child \leftarrow$ random non-terminal
        else
            $child \leftarrow$ random terminal or non-terminal
        end if
        GENERATESUBTREE($remainingTreeDepth - 1$, $child$, $generation\_method$)
    end for
end function
```

Using the tree representation for an individual, the crossover operation can be interpreted as a combination of two trees. The following three steps for crossover are:

1. Select one sub-tree randomly from the first tree.
2. Select a second sub-tree from the second tree with the same return type as the one selected in step 1.
3. Create a new tree with the structure from the first tree. Then replace the selected sub-tree of step 1 with the sub-tree selected in step 2.

Figure 4.2 shows an example of crossover in STGP. The numbers at the trees are corresponding to the previously mentioned three steps. In the first step of the example's crossover operation, a *distance* node of the first tree is selected. This *distance* node returns a double value. Thus, the second tree is searched for a corresponding node with a double value as return type. In the second step, the *subtraction* node is selected. Other possible nodes were all child nodes of the second tree except v_3 and v_4, as they all have a matching return type. The third step then creates a new child-tree with the main structure of the first tree and the replaced sub-tree of the second tree.

Similarly, the mutation of a single individual can be described by the following three steps.

1. Randomly select a sub-tree of the individual for replacement.
2. Generate a valid new sub-tree with the same return type as the old sub-tree similar as described in Algorithm 4.1.
3. Replace the selected sub-tree with the newly generated one.

To prevent the trees from growing too much, different constraints, such as a maximum depth of trees are defined. The constraint to a maximum depth is especially important after mutation or crossover as the trees can grow deeper over time. Thus, the evaluation of a tree would increase. Montana (1995) proposed to return the parents or nothing if a tree gets too deep. Similarly, a parent or nothing is returned if there are no matching sub-trees available during crossover.

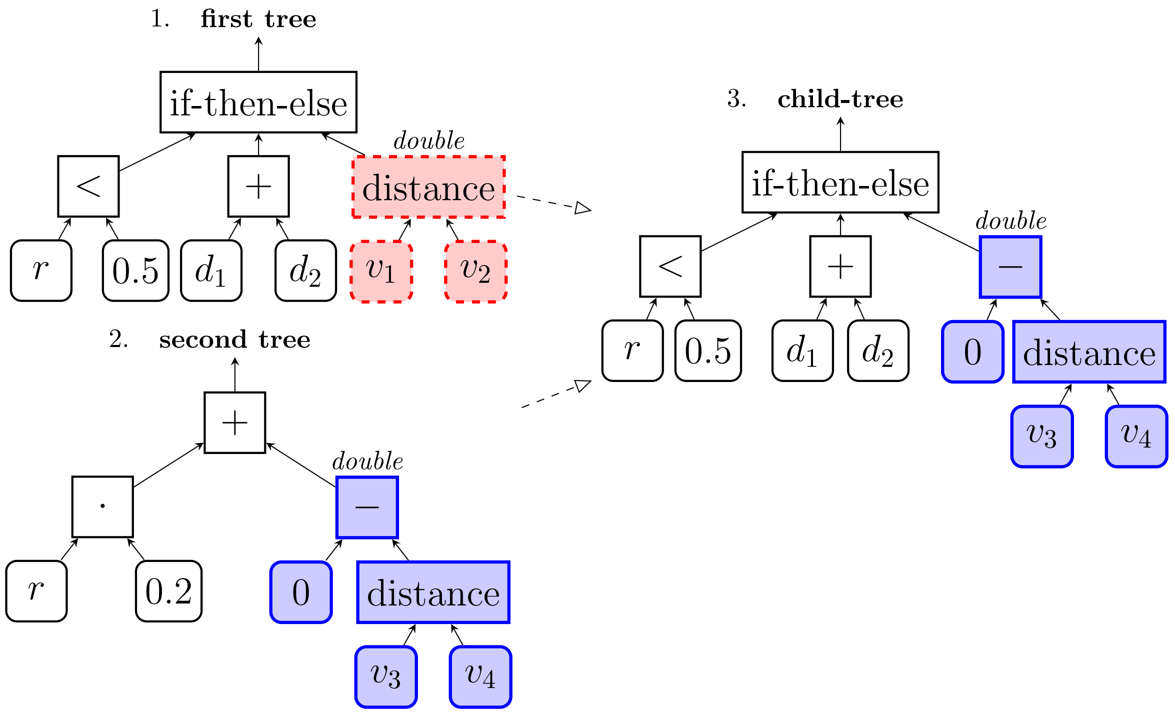


Figure 4.2: Example for crossover in STGP

Different existing frameworks for Java, such as ECJ (Luke *et al.*, 2015), use a similar representation for their individuals. These frameworks are powerful and expandable, but many of them are using reflection and are thus slow. Albeit, STGP is a domain-independent approach, additional domain-specific nodes can be incorporated to represent additional features of a game. These changes are described in Subsection 4.3.2.

4.3 Representation of Individuals in the GVG-AI Competition

4.3.1 (1 + 1) Evolutionary Algorithm

The original implementation of the (1 + 1) EA by Droste *et al.* (2002) used a simple representation where each individual is represented by a Boolean vector. Each Boolean variable influences one coefficient of a function that should be optimized. In the GVG-AI Competition, Perez *et al.* (2014) uses the distances from the avatar to different types of game objects, such as NPCs or walls, as coefficients. A simple Boolean variable could then determine whether a distance is important for optimization or not. As maximizing a linear function with only positive coefficients would be trivial, the Boolean variables have to be adapted to Double variables. These Double variables can then describe the positive or negative weight of each distance feature. Furthermore, the individual should be able to determine the relative strength of an action depending on a given set of features. To be able to incorporate different features based on actions, Lucas, Samothrakis, and Perez (2014), Perez *et al.* (2014) use a Double matrix as an individual instead of a Boolean vector. Each action is represented by a row and each feature by a column. The relative strength of one action can then be described by Equation (4.1).

$$\text{relativeStrength}(a_i) = \sum_{j=1}^{n} w_{ij} \cdot f_j \tag{4.1}$$

One of the n features f_j can be, for instance, the distance to the closest NPC. For each action a_i, a weight w_{ij} for each feature is stored inside the matrix. The weight w_{ij} influences the importance of a feature depending on the given action. The initial values for these weights are normally-distributed random values with mean 0.0 and standard deviation 1.0. These weights are then randomly changed over time by using mutation. One possible mutation operator is described by Droste *et al.* (2002) for Boolean vectors and can be adapted to Double matrix. Droste *et al.* (2002) flips each Boolean value with the probability of $\frac{1}{n}$, where n is the number of features. Accordingly, for each action, each Double weight can be modified with the same probability.

Perez *et al.* (2014) use an additional softmax function to decide which action to choose based on the score obtained by the individual. Applying the softmax function to the weighted sum, a probability for choosing an action is obtained. This thesis uses the alternative approach of choosing an action with an ϵ-Greedy implementation.

One major challenge is that the extracted features can change. There can be, for example, different feature sets for different games. Games that have NPCs can use the distances to the NPCs as a feature. Other games can have another number of NPCs or even none. Furthermore, the number of features can change over time. NPCs can, for instance, spawn on the map or be killed after applying an action. Thus, the EA implementation has to adapt to these circumstances. Perez *et al.* (2014) for example stores the features extracted from a game state. Features are then updated online, so that newly discovered features are added and old ones are removed.

In general, the use of additional features can increase the performance (Lucas *et al.*, 2014) of an EA. Therefore, all distances that can be obtained by using the forward model are used as atomic features. Thus, additional characteristics of the GVG-AI Competition, such as the performance of the framework, can have an impact on the EA. The experiments in Chapter 5 show how well the approach of the (1 + 1) EA with its limited population size performs in the GVG-AI Competition.

4.3.2 Strongly Typed Genetic Programming

The individuals in STGP are represented by trees. The main idea for the GVG-AI Competition is to use each of these trees as evaluation function for a certain state. Based on this evaluation function, an action can be chosen depending on a state of a game. Section 4.4 shows how these functions can be learned inside MCTS.

To be able to build the trees for state evaluation, sets of terminal and non-terminal nodes have to be defined. Some of these nodes are mostly domain-independent, others are added specifically for the GVG-AI Competition. Table 4.1 shows a list of domain-independent nodes that can be used to build trees.

These domain-independent nodes represent basic operations or constant values. Nodes, such as the `LeqNode`, can be used to compare values. The results of this comparison can be used by nodes, such as the `AndNode` or the `OrNode` to combine these values. Other nodes create constants, such as the random but constant double values created by the `RandomConstantNode`. These constants can be used for comparisons or calculations performed by nodes, such as the `PlusNode`. Though these nodes are considered as domain-independent, not all nodes are applicable for all domains. The `Distance2dNode` and the `PlusNode` can, for instance, not be used in a domain where only Boolean features exist.

So far, only domain-independent nodes are available to generate trees. These trees have no option to incorporate information about the state of a game. Thus additional domain dependent nodes have to be implemented. Table 4.2 lists implemented nodes that incorporate additional information for the GVG-AI Competition.

These domain-specific nodes incorporate information provided by the framework of the GVG-AI Competition. Some nodes return positions of different objects, such as the avatar or other game objects. Though these nodes are more specific than the domain-independent ones, they are applicable for many games. Additional action specific nodes incorporate values based on an actions. This allows an individual to calculate action-based scores by using nodes, such as the `ActionWeightNode`. Combined with domain-independent nodes, more complex features of games, such as the distance between the avatar and an NPC, can be built. Figure 4.3 shows an example of a tree that can be generated by using all available nodes.

This example of a tree returns a score based on the number of NPCs. If there are no NPCs left, a high score will be returned. Otherwise, the distance between the avatar and the closest NPC will be returned as the score. The given example is rather simple. The depth is limited and not all information, such as the action that has to be evaluated, are used. Still, it already indicates the idea of creating more complex

Type	Node	Input Types	Return Type	Description
Non-Terminal	`AndNode, NAndNode`	1. Boolean 2. Boolean	Boolean	Logical "And" and "not And" of input parameters.
	`OrNode, NOrNode`	1. Boolean 2. Boolean	Boolean	Logical "Or" and "not Or" of input parameters.
	`NotNode`	1. Boolean	Boolean	Logical "Not" of input parameter.
	`LeqNode`	1. Double 2. Double	Boolean	Returns true if the first input parameter is smaller of equal to the second one.
	`Distance2dNode`	1. Vector2d 2. Vector2d	Double	Calculates the Euclidean distance of the two input vectors.
	`IfThenElseNode`	1. Boolean 2. Double 3. Double	Double	If the first input parameter is true, the second input parameter is returned. Otherwise, the third input parameter is returned.
	`MinusNode`	1. Double 2. Double	Double	Subtraction of both input parameters.
	`PlusNode`	1. Double 2. Double	Double	Addition of both input parameters.
	`MultNode`	1. Double 2. Double	Double	Multiplication of both input parameters.
	`ScaleVector2dNode`	1. Double 2. Vector2d	Vector2d	Scales a two dimensional vector by a Double value.
Terminal	`FalseNode`		Boolean	Returns *false*.
	`TrueNode`		Boolean	Returns *true*.
	`RandomConstantNode`		Double	Returns a randomly chosen, but constant value.
	`OneNode`		Double	Returns 1.
	`ZeroNode`		Double	Returns 0.

Table 4.1: Domain-independent Nodes for STGP in GVG-AI Competition

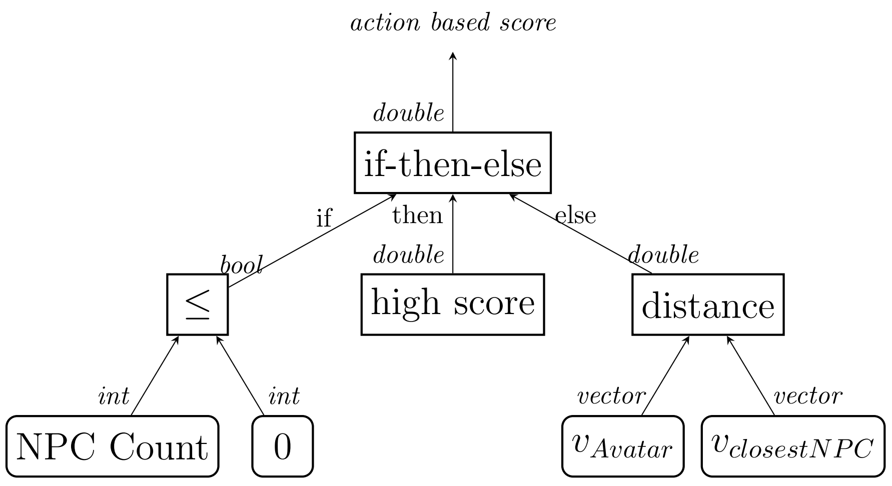


Figure 4.3: Example tree representation of an individual for the GVG-AI Competition

Type	**Node**	**Return Type**	**Description**
Terminal	`ActionWeightNode`	Double	Returns a randomly chosen, but constant value depending on an action.
	`AvatarPositionNode`	Vector2d	Gets the current position of the Avatar.
	`GlobalMaxScoreNode`	Double	Returns the global maximum of scores in an MCTS node obtained so far.
	`GlobalMinScoreNode`	Double	Returns the global minimum of scores in an MCTS node obtained so far.
	`ClosestGameObject -PositionNode`	Vector2d	Returns the closest game object of a certain type form the avatar's position. The following nodes for game objects are implemented: `ClosestImmovablePositionNode` `ClosestMovablePositionNode` `ClosestNpcPositionNode` `ClosestPortalPositionNode` `ClosestResourcesPositionNode`
	`CountGameObjectNode`	Double	Returns the number of a game object with a certain type. The following nodes for game objects are implemented: `CountImmovablePositionsNode` `CountMovablePositionsNode` `CountNpcNode` `CountPortalPositionsNode` `CountResourcePositionsNode`
	`RandomConstant -ActionNode`	Action	Returns a randomly chosen constant action.

Table 4.2: Domain-specific nodes for STGP in the GVG-AI Competition

rules for choosing actions based on a score. Using action-based nodes, such as the `ActionWeightNode`, allows to generate trees that calculate action-based scores. This score can then be embedded in MCTS.

4.4 Embedding Evolution in MCTS Play-out

Both approaches, the (1 + 1) EA and STGP, can give an indication to decide which action to take based on the current state of a game. Similarly to MAST or NST, this score can be used to guide the play-out of MCTS. The advantage of the EA is that it can adapt to different domains, such as different games with different characteristics.

Algorithm 4.2 shows the overall process of one play-out using the (1 + 1) EA as proposed by Perez *et al.* (2014). Additionally, the algorithm shows the simple fitness evaluation and selection of the two individuals.

Algorithm 4.2 MCTS play-out guided by (1 + 1) EA according to Perez *et al.* (2014)

```
Require: $state$: state added during expansion
 1: $individual \leftarrow bestIndividual.\text{COPY}()$
 2: $individual.\text{MUTATE}()$
 3: while $!state.\text{ISGAMEOVER}()$ && $depth < maxSimulationDepth$ do
 4:     if $random.\text{NEXTDOUBLE}() < \epsilon$ then
 5:         $action \leftarrow$ random action            ▷ choose random action with probability of $\epsilon$
 6:     else
 7:         $individual.\text{UPDATEFEATURES}(state)$
 8:         $action \leftarrow individual.\text{GETACTION}(state)$            ▷ choose action based on EA
 9:     end if
10:     $state.\text{ADVANCE}(action)$
11: end while
12: $individual.fitness \leftarrow \text{SCOREPLAYOUT}(state, playoutDepth)$
13: if $individual.fitness > bestIndividual.fitness$ then
14:     $bestIndividual \leftarrow individual$
15: end if
```

At the beginning of each play-out, a new individual is created by copying and mutating the best individual. Then, the state of the game is advanced until the game is over or a maximum simulation depth is reached. With the probability of ϵ, a random action is chosen. Otherwise, the individual is used to calculate values for all legal actions for the current state of the game. Based on these values, the action is chosen greedily and used to advance the state by using the forward model of the GVG-AI Competition.

After the play-out is finished, the fitness of the new individual is set.The heuristics used during the backpropagation described in Subsection 3.3.3 can be applied. Thus, different aspects after the play-out, such as the depth or the score of the last state, can be incorporated. The individual with the highest fitness survives and becomes the next best individual.

This approach can be extended in different ways. In the approach by Perez *et al.* (2014), a population size of two with a 50% survival rate has been chosen, but other configurations are possible. The individual can be used in different steps of MCTS as well, especially in the selection step as suggested by Perez *et al.* (2014). Nevertheless, Perez *et al.* (2014) is only using the individual to guide the play-out. Initializing the individual at the beginning of MCTS allows the individual to guide the selection as well as the play-out. Furthermore, individuals can have different underlying structures, such as the mentioned feature vector representation or the tree structure of STGP. Chapter 5 presents some of these configurations to get an indication of their behavior in the GVG-AI Competition.

A different approach to embed the EA in MCTS is proposed by Benbassat and Sipper (2014). The major difference is that Benbassat and Sipper (2014) allow a larger population size and that STGP is used. The generation of individuals is performed differently, as well as the survival of individuals. For the generation of new individuals, Benbassat and Sipper (2014) propose the so-called "selective crossover" described in Algorithm 4.3.

Algorithm 4.3 Generation of new individuals by selective crossover (Benbassat and Sipper, 2014)

```
1: $parent1, parent2 \leftarrow$ randomly chosen individuals from population for crossover
2: if $parent1.fitness \geq parent2.fitness$ then
3:     $child \leftarrow$ perform one-way crossover with parent1 as donor and parent2 as receiver
4: else
5:     $child1, child2 \leftarrow$ perform two-way crossover
6: end if
7: add children for evaluation
```

The idea which embodies selective crossover is that the fittest individual is more likely to survive. At the beginning of the crossover, two parents are chosen randomly out of the population. Each parent is selected only once per generation for crossover. As the parents are selected individually, there is a 50% that the first selected parent is fitter than the second one. Thus, a one-way crossover is performed in about 50% of all crossover operations. Otherwise, a two-way crossover is performed. Both crossover operations are performed by combining the individuals as described in Subsection 4.2.2. The difference between one-way and two-way crossover is the number of parents that donate a sub-tree or receive one. During one-way crossover, the fitter parent passes a sub-tree to the less fit parent. Thus, the fitter parent survives in 50% of the operations due to the random selection of parents. During two-way crossover the selected sub-trees of both parents are changed.

The created individuals can be evaluated during the play-out. Each play-out determines the fitness of one individual as described in Algorithm 4.4.

Algorithm 4.4 MCTS Play-out according to Benbassat and Sipper (2014)

```
Require: $state$: state added during expansion
Require: $actionEvaluation$: number of random actions that will be evaluated for one step of a play-out
1: $initialScore \leftarrow \textsc{scorePlayout}(state)$
2: $individual \leftarrow \textsc{getNextIndividual}()$
3: while $!state.\textsc{isGameOver}()$ && $depth < maxPlayOutDepth$ do
4:     $bestScore \leftarrow -\infty$
5:     $bestAction \leftarrow ACTIONS.ACTION\_NIL$
6:     $availableActions \leftarrow state.\textsc{getAvailableActions}()$
7:     $\{A\} \leftarrow \textsc{generateRandomSubset}(actionEvaluation, availableActions)$
8:     for all $randomAction : \{A\}$ do
9:         $score \leftarrow individual.\textsc{eval}(state, randomAction)$
10:        if $score > bestScore$ then
11:            $bestScore \leftarrow score$
12:            $bestAction \leftarrow randomAction$
13:        end if
14:    end for
15:    $state.\textsc{advance}(bestAction)$
16: end while
17: $endScore \leftarrow \textsc{scorePlayout}(state)$
18: $individual.fitness \leftarrow endScore - initialScore$
```

At the beginning of each play-out, the score for the current state of the game is calculated with the same strategies described in Subsection 3.3.3. This initial score is later used for fitness evaluation. Then, a previously unevaluated individual is selected. This individual determines which action to take, until the game is over or a maximum play-out depth is reached. The parameter `actionEvaluation` limits the number of different actions that are evaluated at each step. The action with the highest score of the randomly chosen actions is performed. Thus, smaller values for the `actionEvaluation` allow actions with lower scores to be executed. After the play-out is finished, the score of the last state of the play-out is calculated. The fitness of the individual is determined by the gain or loss compared to the initial score.

After all individuals of one population are evaluated during the play-out, a new parent population has to be generated. This process is divided into two steps, the selection step and the procreation step. Algorithm 4.5 describes the selection to select the fittest parents of a population.

Algorithm 4.5 Selection according to Benbassat and Sipper (2014)

```
Require: $tourSize$: number of individuals to be randomly selected from existing population
Require: $winTourSize$: number of individuals to select fittest individuals
Require: $p_{\mathrm{xo}}$: crossover probability
Require: $p_{\mathrm{m}}$: mutation probability
1: $selectedParentCount \leftarrow 0$                                ▷ Begin of Selection
2: repeat
3:     $randomIndividuals \leftarrow tourSize$ randomly chosen, different individuals from population
4:     $bestIndividuals \leftarrow$ best $winTourSize$ individuals from $randomIndividuals$
5:     add copy of $bestIndividuals$ to next parent population
6:     $selectedParentCount \leftarrow selectedParentCount + winTourSize$
7: until $selectedParentCount == initialPopulationSize$
8: CROSSOVER(next parent population, $p_{\mathrm{xo}}$)                ▷ Begin of Procreation
9: MUTATION(next parent population, $p_{\mathrm{m}}$)
```

During the selection step, a random subset of the population is selected. Each individual can be selected multiple times over multiple iterations, but only once during one iteration. The fittest individuals of this subset are selected for the next parent population. This process is repeated until the size of the newly generated population is equal to the size of the old population. Individuals are selected by using random components and by using their fitness. Less fitter individuals can be selected due to the first random selection. Yet, fitter individuals are more likely to survive because of the second selection. This behavior can be tuned with the parameters `TourSize` and `WinTourSize`. Smaller values for the `TourSize`, for example, increase the probability of less fitter individuals to survive as fewer individuals are selected for the first subset. After the individuals for the new parent population are selected, the procreation step starts. With the probability p_{xo}, two individuals are chosen for crossover as described in Algorithm 4.3. After the crossover is finished, each individual is mutated with the probability of p_{m}. Using the procreation approach of Benbassat and Sipper (2014), each individual can be chosen for crossover and mutation.

As the time for each play-out is limited by the GVG-AI Competition, the number of individuals that can be evaluated is reduced. Depending on the population size, it is even possible that less than one population can be evaluated during one game cycle. Thus, the population and the fitness of all individuals are stored between different game cycles. As the time for initializing the agent in the GVG-AI Competition is higher than for choosing one action, the time for initialization can be used to create and improve an initial population. After a random initial population is created, the remaining time is used to perform some iterations of MCTS to evaluate the fitness of individuals. This could improve the overall performance of the EA, as more generations can be created.

Chapter 5

Experiments and Results

Using the previously described strategies and combinations of those, many agents for the competitions can be built. This chapter describes experiments to test different aspects of the strategies and the GVG-AI Competition to increase the agents' performance. The test setup is described, the results of different tests are presented and finally evaluated to answer the research questions.

Chapter contents: Test Setup, Results & Evaluation

5.1 Test Setup

5.1.1 Overview

The framework of the GVG-AI Competition is developed in Java and provides all classes and allows to modify them. To test the agents' performance, this framework has to be extended. The extensions allow gathering and writing additional statistics, such as the number of performed iterations of MCTS. These changes to the framework have low impact on the agents' performance. Particularly time consuming actions, like the writing of files, can be performed between moves. Thus the agents have the same amount of available time in the test setup as during the real competition.

The gathered statistics are obtained by the forward model of the GVG-AI Competition as well as the different strategies for MCTS steps. The forward model provides information, such as the winner of a game, the score or the number of turns that were played during one game. The different strategies for MCTS, such as UCT selection or NST play-out, can have additional interesting aspects for testing. Depending on different influences, such as a score range or a game, the UCT selection can have a change in performance depending on the parameter C. The NST play-out has additional parameters, such as the number of actions for the N-grams or a factor to decay scores. All parameters are written to a CSV file for evaluation. In addition to the tuning of parameters, the developed agent allows combining different strategies for each step of MCTS.

The results are tested against the existing three training sets, the two training sets of 2014 and 2015 and the validation set of 2014 from the competition. Each set contains ten games with five levels for each game. For most tests of this thesis, each level is performed ten times. Thus, one test run contains $3 \cdot 10 \cdot 5 \cdot 10 = 1500$ played games, similar to the test setup by Perez *et al.* (2015). To test different game specific aspects, the test is repeated several times. All tests are based on samples sizes of at least 50 played games for each game. Most tests are even based on 500 samples for a game set, 1500 samples for all game sets or even more due to repetition of the tests. Thus, a standard score is used to calculate the confidence bounds. A 95% confidence bound is chosen to estimate the significance of the results presented in this chapter.

Since the beginning of the GVG-AI Competition, the framework has been extended and changed. Thus, it is difficult to compare the results of this thesis to existing results, such as the results by Perez *et al.* (2015). The state of the framework that is used for all tests presented in the following sections was taken at March 14, 2015 (GitHub, 2015a).

5.1.2 Hardware Specifications

The experiments for this thesis are performed on a Linux cluster of the Department of Knowledge Engineering of Maastricht University. The cluster has one master node with ten AMD nodes and two Intel nodes. Only the AMD nodes are used to execute the tests and generate data. The different hardware components are specified in Table 5.1.

Node	Component	Specifications
Master	CPU	2 × AMD Dual-Core Opteron F 2220, 2.8 GHz, 95 Watt (max. 2 Opteron Socket F processors)
	RAM	8 GB DDR2 DIMM, reg. ECC (4 DIMMs, max. 16 DIMMs)
	Graphics	NVIDIA nForce MCP55 Pro, NEC uPD720400 chipset
	Hard Disk	2 × 80 GB hot-swap SATA-2 hard disk, 7200 rmp, 3.5”, max. 8 hot-swap hard disks, Configuration of a RAID system as RAID level 1 (Mirroring) 5 × 500 GB hot-swap SATA-2 hard disk, 7200 rmp, 3.5”, max. 8 hot-swap hard disks, Configuration of a RAID system as RAID level 5 (Parity)
	RAID-controller	3Ware SATA-2 RAID-controller, 8-ports, PCI-X LP
AMD	CPU	2 × AMD Dual-Core Opteron F 2216, 2.4 GHz, 95 Watt (max. 2 Opteron Socket F Processors)
	RAM	8 GB DDR2 DIMM, reg. ECC (4 DIMMs, max. 32 GB, 16 DIMMs)
	Graphics	NVIDIA nForce4 2200 Professional chipset
	Hard Disk	80 GB hot-swap SATA hard disk, max. 2 hot-swap hard disks

Table 5.1: Hardware specifications of nodes used for testing

The AMD nodes are used to run the Java application for testing the agents' performance. The tests are performed with the Java(TM) SE Runtime Environment (build 1.7.0_40-b43) and a Java HotSpot(TM) 64-Bit Server VM (build 24.0-b56, mixed mode).

5.2 Results

5.2.1 Comparability to Sample OL-MCTS Player

Different aspects of agents can have impact on their performance. One aspect is the performance of the agent with regards to software engineering techniques. The implemented framework to test different strategies uses multiple interfaces and statistic classes to get additional test results. This framework uses the time management of the GVG-AI Competition and a similar overall structure due to the concept and steps of MCTS as described in Subsection 3.4.2 . To compare these influences to the sample agents, a custom implementation of the sample OL-MCTS player is used. The corresponding strategies in the framework for this thesis are an “Open Loop” UCT selection, with a random expansion and a random play-out. The sample OL-MCTS player uses $C = \sqrt{2}$ for UCT selection. With the same value for C, the custom implementation won in 23.1 ± 1.51 percent of all games in all game sets. Compared to the 22.77 ± 1.50 percent of the sample OL-MCTS player, this is a small, insignificant difference. Yet it indicates that both implementations are comparable and perform similarly. Even looking at different game sets or games, the two implementations show similar behavior. Thus, the sample OL-MCTS player can be seen as starting point for improvement. Different strategies are added and tuned during the next sections to evaluate the improvements.

5.2.2 Game Characteristics & Sample Agents

MCTS based agents improve their results while growing the search tree. The larger the amount of time to decide which action to take, the more play-outs can be performed. But the available time is not the only influence on the number of iterations of MCTS. A good play-out strategy, for example, can shorten each play-out. Thus, more play-outs can be performed. Another important aspect of the GVG-AI Competition

		Avg. Iterations per Game Cycle			Win Percentage		
Game	**$\|A\|$**	**MCTS**	**OL-MCTS**	**NST**	**MCTS**	**OL-MCTS**	**NST**
Aliens	3	$28.3 \pm 1.0\,[1, 290]$	$39.3 \pm 0.7\,[1, 210]$	$33.9 \pm 0.9\,[1, 169]$	100.0 ± 0.0	99.0 ± 2.0	100.0 ± 0.0
Boulderdash	5	$6.9 \pm 0.3\,[1, 57]$	$8.8 \pm 0.2\,[1, 47]$	$8.3 \pm 0.3\,[1, 37]$	2.0 ± 2.7	3.0 ± 3.3	3.0 ± 3.3
Butterflies	4	$18.9 \pm 1.1\,[1, 68]$	$30.8 \pm 0.9\,[14, 108]$	$35.5 \pm 1.6\,[11, 99]$	92.0 ± 5.3	88.0 ± 6.4	96.0 ± 3.8
Chase	4	$12.9 \pm 0.7\,[1, 89]$	$21.5 \pm 0.7\,[15, 75]$	$22.2 \pm 0.9\,[1, 59]$	0.0 ± 0.0	2.0 ± 2.7	0.0 ± 0.0
Frogs	4	$22.2 \pm 1.3\,[1, 60]$	$25.2 \pm 1.3\,[12, 52]$	$15.9 \pm 0.8\,[6, 32]$	4.0 ± 3.8	7.0 ± 5.0	13.0 ± 6.6
Missile Command	5	$53.2 \pm 1.3\,[34, 582]$	$58.1 \pm 0.9\,[41, 304]$	$47.7 \pm 1.7\,[12, 250]$	52.0 ± 9.8	51.0 ± 9.8	60.0 ± 9.6
Portals	4	$14.3 \pm 0.7\,[1, 80]$	$23.5 \pm 0.9\,[21, 72]$	$23.4 \pm 0.9\,[13, 55]$	15.0 ± 7.0	17.0 ± 7.4	15.0 ± 7.0
Sokoban	4	$11.5 \pm 1.6\,[1, 132]$	$23.1 \pm 2.3\,[1, 148]$	$24.9 \pm 1.9\,[9, 105]$	5.0 ± 4.3	14.0 ± 6.8	17.0 ± 7.4
Survive Zombies	4	$12.1 \pm 0.6\,[4, 113]$	$17.5 \pm 0.6\,[10, 77]$	$16.7 \pm 0.7\,[1, 61]$	39.0 ± 9.6	40.0 ± 9.6	47.0 ± 9.8
Zelda	5	$20.2 \pm 1.7\,[1, 298]$	$30.7 \pm 2.1\,[1, 212]$	$30.3 \pm 1.7\,[1, 176]$	5.0 ± 4.3	11.0 ± 6.1	17.0 ± 7.4
2014 Training Set		$\mathbf{20.0 \pm 0.8\ [1, 582]}$	$\mathbf{27.9 \pm 0.9\ [1, 304]}$	$\mathbf{25.9 \pm 0.8\ [1, 250]}$	$\mathbf{31.4 \pm 2.9}$	$\mathbf{33.2 \pm 2.9}$	$\mathbf{36.8 \pm 3.0}$
Camel Race	4	$29.1 \pm 2.0\,[1, 232]$	$36.8 \pm 1.8\,[24, 123]$	$34.1 \pm 1.7\,[11, 105]$	4.0 ± 3.8	5.0 ± 4.3	6.0 ± 4.7
Digdug	5	$5.0 \pm 0.3\,[1, 58]$	$8.3 \pm 0.2\,[1, 51]$	$8.1 \pm 0.3\,[3, 40]$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Eggomania	3	$31.9 \pm 0.8\,[1, 264]$	$34.9 \pm 0.7\,[26, 117]$	$30.6 \pm 1.1\,[1, 106]$	4.0 ± 3.8	3.0 ± 3.3	11.0 ± 6.1
Firecaster	5	$10.5 \pm 1.0\,[1, 185]$	$16.7 \pm 1.0\,[10, 110]$	$17.4 \pm 0.9\,[3, 100]$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Firestorms	4	$8.2 \pm 0.2\,[1, 52]$	$11.7 \pm 0.5\,[6, 47]$	$11.2 \pm 0.7\,[5, 40]$	2.0 ± 2.7	7.0 ± 5.0	12.0 ± 6.4
Infection	5	$11.7 \pm 0.5\,[1, 180]$	$17.6 \pm 0.5\,[3, 47]$	$17.3 \pm 0.6\,[4, 95]$	81.0 ± 7.7	78.0 ± 8.1	79.0 ± 8.0
Overload	5	$10.3 \pm 1.3\,[1, 222]$	$17.5 \pm 1.7\,[1, 170]$	$18.8 \pm 1.3\,[8, 102]$	16.0 ± 7.2	16.0 ± 7.2	14.0 ± 6.8
Pac-Man	4	$2.5 \pm 0.2\,[1, 16]$	$5.7 \pm 0.2\,[1, 18]$	$6.3 \pm 0.3\,[1, 20]$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Seaquest	5	$13.5 \pm 1.1\,[5, 327]$	$20.8 \pm 1.2\,[12, 93]$	$18.8 \pm 1.0\,[8, 182]$	41.0 ± 9.6	45.0 ± 9.8	63.0 ± 9.5
Whac-A-Mole	4	$43.2 \pm 0.9\,[30, 411]$	$59.8 \pm 1.2\,[42, 211]$	$56.4 \pm 2.4\,[28, 200]$	81.0 ± 7.7	97.0 ± 3.3	97.0 ± 3.3
2014 Validation Set		$\mathbf{16.6 \pm 0.8\ [1, 411]}$	$\mathbf{23.0 \pm 1.0\ [1, 211]}$	$\mathbf{21.9 \pm 1.0\ [1, 200]}$	$\mathbf{22.9 \pm 2.6}$	$\mathbf{25.1 \pm 2.7}$	$\mathbf{28.2 \pm 2.8}$
Bait	4	$9.2 \pm 2.4\,[1, 287]$	$16.7 \pm 2.8\,[1, 238]$	$17.2 \pm 2.1\,[1, 74]$	6.0 ± 4.7	5.0 ± 4.3	4.0 ± 3.8
Bolo Adventures	4	$3.1 \pm 0.3\,[1, 49]$	$6.3 \pm 0.4\,[1, 40]$	$6.7 \pm 0.5\,[1, 33]$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Brain Man	4	$9.0 \pm 1.4\,[1, 168]$	$15.7 \pm 1.5\,[8, 123]$	$15.3 \pm 1.3\,[6, 120]$	4.0 ± 3.8	1.0 ± 2.0	2.0 ± 2.7
Chips Challenge	4	$7.6 \pm 0.7\,[1, 89]$	$12.5 \pm 0.8\,[1, 60]$	$12.0 \pm 0.7\,[1, 57]$	8.0 ± 5.3	7.0 ± 5.0	0.0 ± 0.0
Modality	4	$9.2 \pm 2.8\,[1, 77]$	$11.2 \pm 2.1\,[1, 54]$	$12.5 \pm 2.2\,[1, 97]$	25.0 ± 8.5	23.0 ± 8.2	29.0 ± 8.9
Painter	4	$6.5 \pm 1.4\,[1, 129]$	$11.1 \pm 1.6\,[1, 107]$	$9.7 \pm 0.8\,[1, 152]$	52.0 ± 9.8	43.0 ± 9.7	45.0 ± 9.8
Real Portals	5	$6.8 \pm 0.5\,[1, 77]$	$9.8 \pm 0.6\,[7, 62]$	$9.9 \pm 0.6\,[1, 62]$	0.0 ± 0.0	1.0 ± 2.0	0.0 ± 0.0
Real Sokoban	4	$9.4 \pm 1.3\,[1, 109]$	$19.5 \pm 1.8\,[14, 132]$	$21.3 \pm 1.6\,[1, 114]$	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
The Citadel	4	$5.4 \pm 0.8\,[1, 132]$	$9.4 \pm 0.7\,[6, 88]$	$9.9 \pm 0.7\,[1, 70]$	10.0 ± 5.9	5.0 ± 4.3	5.0 ± 4.3
Zen Puzzle	4	$8.5 \pm 0.6\,[1, 115]$	$17.2 \pm 1.1\,[13, 133]$	$24.0 \pm 1.9\,[12, 102]$	8.0 ± 5.3	15.0 ± 7.0	2.0 ± 2.7
2015 Training Set		$\mathbf{7.5 \pm 0.5\ [1, 287]}$	$\mathbf{12.9 \pm 0.5\ [1, 238]}$	$\mathbf{13.8 \pm 0.5\ [1, 152]}$	$\mathbf{11.3 \pm 2.0}$	$\mathbf{10.0 \pm 1.9}$	$\mathbf{8.7 \pm 1.7}$
Grand Total		$\mathbf{14.7 \pm 0.5\ [1, 582]}$	$\mathbf{21.3 \pm 0.5\ [1, 304]}$	$\mathbf{20.5 \pm 0.5\ [1, 250]}$	$\mathbf{21.9 \pm 1.5}$	$\mathbf{22.8 \pm 1.5}$	$\mathbf{24.6 \pm 1.5}$

Table 5.2: Number of MCTS iterations per game and win percentages for sample MCTS, OL-MCTS and NST agents

is the framework itself. The games have many different characteristics, for instance, a varying number of NPCs or different map sizes. All these characteristics can influence the performance of components, such as the forward model. Table 5.2 shows the number of available actions for each game and the performance of the two sample MCTS agents of the GVG-AI Competition and a NST implementation for this thesis. The NST-based agent uses 3-grams, chooses actions randomly with a probability of $\epsilon = 0.5$ and decays the statistics with a decaying factor of 0.15. Furthermore, the action-based statistics are used for selection with Progressive History (Subsection 3.3.1), where $W = 1$ and $C = 0.6$. The parameters were determined empirically by testing multiple configurations. The performance of the agents is measured by the average number of iterations performed per game, the minimum and maximum performed iterations per game cycle and the win ratio.

The average number of iterations per game cycle shows major differences between each game of the GVG-AI Competition. Overall, the number of iterations decreases for each game set. For some games, it is close to the number of available actions. While playing these games, the search tree of MCTS has a depth close to one and the action selection has to be based on a small number of play-outs. This could be the major reason for the significantly worse performance of all agents playing the third game set.

The number of game objects in a game is related to the number of iterations. At the time of testing, the forward-model of the GVG-AI Competition calculates all available information after an action is simulated. Even when no information about the position of the different game objects is used, all different lists used by the forward model are updated. This takes much time and especially slows down games with a higher number of game objects. *Pac-Man* is, for instance, one of the biggest games and contains many pills on the map. In *Missile Command*, it is the opposite case, as there are only four missiles and three cities. In many games, such as *Pac-Man* or *Aliens*, the minimum of performed iterations is one. In most games, this minimum occurs at the beginning of a game and in the endgame more iterations can be performed. Other frameworks that specialize in a certain game can allow much more iterations. In a Java framework specialized in *Pac-Man*, agents can perform between 200 and 400 iterations every 40 milliseconds (Pepels, Winands, and Lanctot, 2014b). The framework of the GVG-AI Competition allows playing a variety of different games, but thereby loses performance. Even the NST implementation, which performed best in *Pac-Man*, can perform only 1 to 20 iterations per game cycle.

In addition to the information about the game characteristics, the results can give an indication whether to use an "Open Loop" (Perez *et al.*, 2015) implementation or not. The results are similar for both implementations of the sample agents. In most games, the win ratio of the SAMPLE OL-MCTS PLAYER is insignificantly higher compared to the SAMPLE MCTS PLAYER. Regarding the number of iterations, the SAMPLE OL-MCTS PLAYER is able to perform significantly more iterations. This can be explained by the character of the implementation, as no additional game states are stored inside the nodes of MCTS. Furthermore, the game states are copied less in the "Open Loop" implementation. Due to the significant increase of iterations, all future evaluations in this thesis are using the "Open Loop" implementation. In addition to the increased number of iterations, the "Open Loop" approach makes sense, due to the non-deterministic character of some games. Storing a game state inside a node of MCTS poses the disadvantage that the state has to be updated each time the node is visited as the game state could have changed due to random characteristics.

The NST-based agent uses a more complex approach, as the statistics for N-grams of actions are gathered. They are used to guide the selection, the play-out and are decayed after executing an action. Yet, the number of iterations is similar to the SAMPLE OL-MCTS PLAYER. In some games, such as *Zen Puzzle*, NST is able to perform significantly more iterations. One possible explanation is that the action-based statistics guide the play-out to shorter games and thus allow more iterations. The win ratio of the NST agent is insignificantly higher.

Looking at the average number of game cycles performed per game of different agents, it becomes clear that won games tend to be much shorter than lost games. The average number of game cycles for wins is 613, about the half compared to the numbers for the losses with 1381 cycles. In many games, agents lost because of the time limit. In about 230,000 games of different agents performed for this thesis, about 115,000 games were lost due to the limitation of 2000 game cycles. This can have different reasons. In some games, the agent moves objects so that it cannot reach the goal anymore. In other games, the agent simply does not have enough time to solve the game due to, for instance, the horizon effect. As a result, the average game takes longer. Testing one game set where each game is played ten times takes about 6.5 hours of time. Thus, running all three game sets takes about 19.5 hours.

5.2.3 Score Heuristics

The heuristic to determine a score after a play-out has influence on the selection step of MCTS. Figure 5.1 shows the different score heuristics that have been discussed in this thesis. All heuristics are tested with an "Open Loop" UCT selection in combination with a random play-out and a random expansion. As the score heuristics have different ranges for scores, values for $C = \{0.5, 0.6, 0.7\}$ are tested and the best result is presented. Parameters of all heuristics are chosen according to the original implementations. The graph with error bars shows the win percentage, the columns represent the obtained score.

Score Heuristic	**Win Percentage**	**Average Score**
Regular MCTS	19.60 ± 2.01	35.56 ± 8.88
QB without bonus	18.53 ± 1.97	35.41 ± 8.31
QB with qualitative bonus	20.13 ± 2.03	42.84 ± 9.06
QB with relative bonus	20.33 ± 2.04	33.39 ± 7.51
QB with both bonuses	23.93 ± 2.16	43.81 ± 8.87
QB, separated statistics	25.33 ± 2.20	46.97 ± 9.81
Normalized	20.27 ± 2.03	52.84 ± 13.82
GVG-AI	25.33 ± 2.20	49.62 ± 11.41
GVG-AI with depth bonus	25.33 ± 2.20	47.54 ± 9.99

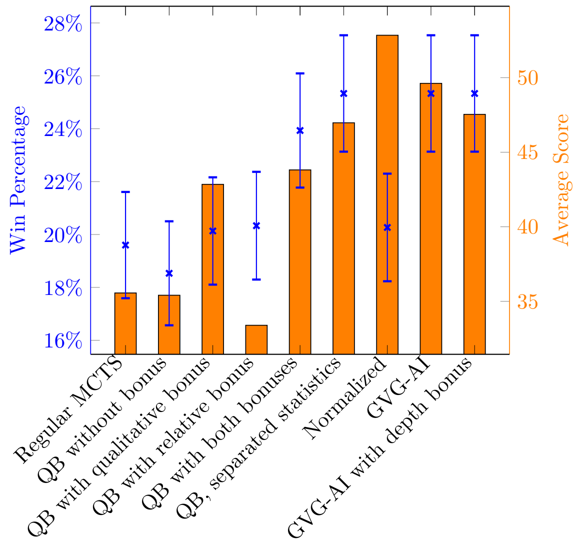


Figure 5.1: Evaluation of score heuristics

The "regular MCTS" (Equation (3.2)) uses only the information about a win or loss of a game. The

Quality-Based (QB) heuristic (Equation (3.9)) is tested with different configurations to get information about the influence of the score and the game length on the score. As this heuristic needs additional statistics and the available time is limited, it is tested without any bonus. In spite of the additional statistics, the quality-based heuristic performs about the same as the "regular MCTS" heuristic. Both heuristics have a low win rate as well as a low average score. Adding a qualitative bonus with $a = 0.25$ insignificantly increases the average score, but has no significant impact on the win ratio. Adding the relative bonus with $a = 0.25$ has no impact on the score and, again, only a low insignificant increase on the win ratio. Using both bonuses increases the score slightly and the win rate significantly compared to the "regular MCTS" heuristic. Using different statistics for wins and losses to calculate the bonuses slightly increases the performance again. Similar to the quality-based heuristic with a qualitative bonus, the "Normalized" heuristic (Equation (3.8)) improves the score and has a similar impact to the win ratio, when compared to regular MCTS.

The handcrafted "GVG-AI" heuristic (Equation (3.7)), which is used by the sample MCTS agents, adds a large bonus or penalty for wins or losses to the score. The heuristic can be extended by a relative bonus, similar to the quality-based heuristic. The idea of this "GVG-AI (heuristic) with depth bonus" is to favor short games in case of a win and otherwise long games. The depth is scaled by a tunable parameter and added to the score and then normalized. The result is similar to the regular GVG-AI heuristic.

In 1500 played games, agents using the GVG-AI heuristics with and without depth bonus and the quality-based heuristic with separate statistics won 380 games. Thus, these three heuristics perform best over all game sets of the GVG-AI Competition. There are only some minor differences between games and other aspects, such as the obtained score.

A single bonus of the quality-based heuristic has not much influence on the win ratio. Combining the "QB with both bonuses" (Equation (3.9)) increases the performance significantly. One reason could be a scaling issue that can be fixed by tuning the parameters. Thus, the parameter a that is used for both bonuses is separated into a_r and a_q. The parameter a_r can be used to tune the game length based relative bonus. The parameter a_q can be used to tune the qualitative bonus that adds bonus based on the score obtained by the forward model. Different combinations of these parameters are tested and then compared to the other approaches. Table 5.3 shows the best results obtained for a score heuristic after tuning the parameters.

The results are in a similar range as before tuning the parameters. The only real improvements are obtained by tuning the parameters of the quality-based approach. Increasing the parameters a to $a_r = 0.9$ and $a_q = 0.6$ and $C = 0.6$ leads to a small improvement. Separating the statistics for wins and losses seems to have less influence. One reason for this behavior could be the short play-out depth, as fewer final game states are reached during play-out. The quality-based heuristic with separated statistics performs best with $a_r = a_q = 0.25$, as suggested by Pepels *et al.* (2014a).

The results between the different games are similar in all implementations. Some games, such as *Aliens* or *Infection*, are played with good, sometimes even perfect results. Other games, especially in the third game set, are only won randomly. Only in some games, like *Zen Puzzle*, the GVG-AI heuristics performed significantly better than the other ones, though the variance is rather high.

The score can reveal different information about a game. In some games, such as *Aliens* or *Butterflies*, killing enemies increases the score. Therefore, a higher score means that the agent is getting closer to winning the game. In other games, the score does not directly correspond to winning the game. In *Painter*, for example, the agent has to paint the initial grey board blue by moving over a cell. Moving over a grid cell changes the color from grey to blue or vice versa. Each color change to blue increases the score, thus a higher score can lead to longer games. These games should favor from a depth-based strategy, but the results show no significant differences. Even regarding other aspects, such as the time that is necessary to end the game or the final score, shows only small changes. The presented techniques are not significantly better compared to the heuristic used by the GVG-AI Competition. Nevertheless, the results show that the score obtained by the forward model is a good indication for the game state. Using this additional information increases the agent's win ratio from about 20% in a regular MCTS implementation to almost 26% by using qualitative and relative bonuses.

Game	QB	QB separated statistics	GVG-AI with Depth	GVG-AI	Normalized	Regular MCTS
Aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	94.0 ± 6.6	66.0 ± 13.1
Boulderdash	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 5.4	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 5.4
Butterflies	96.0 ± 5.4	100.0 ± 0.0	96.0 ± 5.4	90.0 ± 8.3	66.0 ± 13.1	66.0 ± 13.1
Chase	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 5.4	0.0 ± 0.0	0.0 ± 0.0
Frogs	18.0 ± 10.6	24.0 ± 11.8	4.0 ± 5.4	4.0 ± 5.4	4.0 ± 5.4	8.0 ± 7.5
Missile Command	70.0 ± 12.7	56.0 ± 13.8	58.0 ± 13.7	62.0 ± 13.5	40.0 ± 13.6	54.0 ± 13.8
Portals	16.0 ± 10.2	12.0 ± 9.0	6.0 ± 6.6	16.0 ± 10.2	10.0 ± 8.3	14.0 ± 9.6
Sokoban	32.0 ± 12.9	18.0 ± 10.6	8.0 ± 7.5	16.0 ± 10.2	10.0 ± 8.3	6.0 ± 6.6
Survive Zombies	42.0 ± 13.7	44.0 ± 13.8	42.0 ± 13.7	40.0 ± 13.6	40.0 ± 13.6	34.0 ± 13.1
Zelda	16.0 ± 10.2	24.0 ± 11.8	12.0 ± 9.0	14.0 ± 9.6	4.0 ± 5.4	14.0 ± 9.6
2014 Training Set	**39.0 ± 4.3**	**37.8 ± 4.2**	**33.0 ± 4.1**	**34.6 ± 4.2**	**26.8 ± 3.9**	**26.6 ± 3.9**
Camel Race	10.0 ± 8.3	10.0 ± 8.3	8.0 ± 7.5	10.0 ± 8.3	4.0 ± 5.4	10.0 ± 8.3
Digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Eggomania	10.0 ± 8.3	6.0 ± 6.6	6.0 ± 6.6	8.0 ± 7.5	10.0 ± 8.3	4.0 ± 5.4
Firecaster	4.0 ± 5.4	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 5.4	0.0 ± 0.0	0.0 ± 0.0
Firestorms	20.0 ± 11.1	10.0 ± 8.3	6.0 ± 6.6	8.0 ± 7.5	0.0 ± 0.0	6.0 ± 6.6
Infection	82.0 ± 10.6	82.0 ± 10.6	80.0 ± 11.1	82.0 ± 10.6	76.0 ± 11.8	68.0 ± 12.9
Overload	14.0 ± 9.6	32.0 ± 12.9	24.0 ± 11.8	20.0 ± 11.1	16.0 ± 10.2	8.0 ± 7.5
Pac-Man	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.0 ± 3.9	0.0 ± 0.0	0.0 ± 0.0
Seaquest	58.0 ± 13.7	60.0 ± 13.6	42.0 ± 13.7	44.0 ± 13.8	58.0 ± 13.7	42.0 ± 13.7
Whac-A-Mole	100.0 ± 0.0	100.0 ± 0.0	96.0 ± 5.4	96.0 ± 5.4	84.0 ± 10.2	94.0 ± 6.6
2014 Validation Set	**29.8 ± 4.0**	**30.0 ± 4.0**	**26.2 ± 3.9**	**27.4 ± 3.9**	**24.8 ± 3.8**	**23.2 ± 3.7**
Bait	8.0 ± 7.5	4.0 ± 5.4	8.0 ± 7.5	10.0 ± 8.3	2.0 ± 3.9	8.0 ± 7.5
Bolo Adventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Brain Man	2.0 ± 3.9	0.0 ± 0.0	14.0 ± 9.6	8.0 ± 7.5	2.0 ± 3.9	4.0 ± 5.4
Chips Challenge	0.0 ± 0.0	0.0 ± 0.0	16.0 ± 10.2	14.0 ± 9.6	2.0 ± 3.9	0.0 ± 0.0
Modality	22.0 ± 11.5	24.0 ± 11.8	26.0 ± 12.2	24.0 ± 11.8	36.0 ± 13.3	28.0 ± 12.4
Painter	48.0 ± 13.8	48.0 ± 13.8	62.0 ± 13.5	50.0 ± 13.9	40.0 ± 13.6	40.0 ± 13.6
Real Portals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Real Sokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
The Citadel	6.0 ± 6.6	6.0 ± 6.6	12.0 ± 9.0	6.0 ± 6.6	6.0 ± 6.6	8.0 ± 7.5
Zen Puzzle	2.0 ± 3.9	0.0 ± 0.0	30.0 ± 12.7	28.0 ± 12.4	4.0 ± 5.4	2.0 ± 3.9
2015 Training Set	**8.8 ± 2.5**	**8.2 ± 2.4**	**16.8 ± 3.3**	**14.0 ± 3.0**	**9.2 ± 2.5**	**9.0 ± 2.5**
Grand Total	**25.9 ± 2.2**	**25.3 ± 2.2**	**25.3 ± 2.2**	**25.3 ± 2.2**	**20.3 ± 2.0**	**19.6 ± 2.0**

Table 5.3: Win percentages of best score heuristics by game

5.2.4 Selection

The selection step of MCTS can be enhanced in several ways. In Subsections 3.2.1 and 3.3.1, some existing strategies and enhancements are described. Five of these strategies are based on UCT and two are based on a ϵ-Greedy selection. The parameter C of UCT depends on the range of scores for a MCTS node. For example, if the average scores are low, a relatively high value for C can favor exploration. Figure 5.2 shows an overview of the performance of different agents depending on C. All tests in this subsection are performed in combination with a random play-out and a random expansion strategy. The blue graph with error bars represents the win percentage, the orange columns display the obtained score.

Changing C in the custom implementation has no significant impact on the performance. Similar behavior was observed for most tested agents. Only some agents performed insignificantly better for values $C \in \{0.5, 0.6, 0.7\}$. Similar values are used in other agents, such as the EvoMCTS player by Benbassat and Sipper (2014) with $C = 0.7$. Thus, the UCT selection strategies that can be tuned by the parameter C are tested with these values. Similarly, different values of $\epsilon \in \{0.0 \ldots 0.8\}$ are tested. Only the results obtained by the best performing set of parameters are used. Table 5.4 presents the results of the different selection strategies and compares them to a random selection. Again, the blue data points with error bars represent the win percentage and the orange columns display the obtained score. As the differences in results are small, the sample size per selection strategy is based on 3000 played games.

The ϵ-Greedy strategies and the random selection get the lowest number of wins. In general, a greedy selection does not seem to work well in the GVG-AI Competition, as low values for ϵ perform even worse. All of the remaining UCT selection strategies perform better than the ϵ-Greedy and random strategies. The Progressive History only performs insignificantly better than a random selection. All other UCT-based strategies improve the win ratio significantly. The UCB1-Tuned strategy appears to perform best and has the advantage that it has no additional parameter that has to be tuned. This could be an advantage in the GVG-AI Competition, due to the variety of games.

C	**Win Percentage**	**Avg. Score**
0.0	20.9 ± 1.0	43.7 ± 5.1
0.1	20.8 ± 1.0	44.0 ± 5.4
0.2	21.1 ± 1.0	46.0 ± 5.5
0.3	21.7 ± 1.0	43.5 ± 4.9
0.4	22.1 ± 1.1	48.1 ± 5.7
0.5	22.1 ± 1.0	50.9 ± 6.3
0.6	21.5 ± 1.0	47.5 ± 5.7
0.7	21.9 ± 1.0	46.7 ± 5.5
0.8	20.9 ± 1.0	45.2 ± 5.2
0.9	21.5 ± 1.0	42.6 ± 4.7
1.0	21.2 ± 1.0	46.4 ± 5.5
1.1	21.3 ± 1.0	43.6 ± 5.1
1.2	21.1 ± 1.0	42.6 ± 4.9
1.3	21.6 ± 1.0	44.0 ± 5.1
1.4	21.1 ± 1.0	43.5 ± 5.1

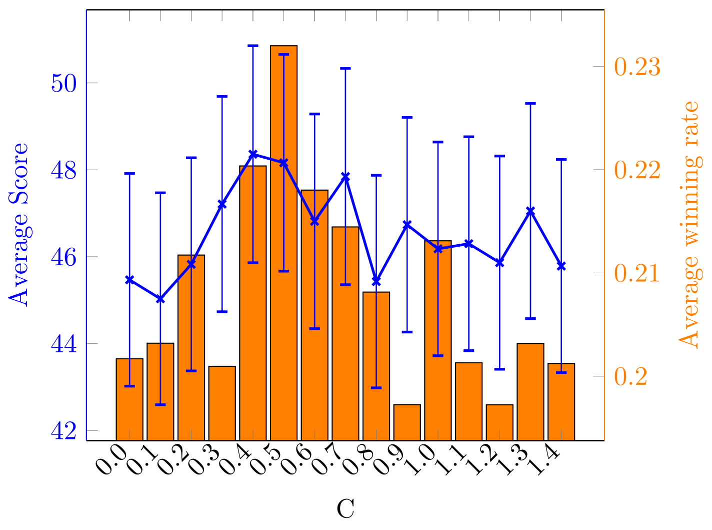


Figure 5.2: Evaluation of UTC's C values of different agents

Selection Strategy	**Win Percentage**	**Average Score**	**Average Iterations per Game Cycle**
UCB1-Tuned (Equation (3.4))	24.80 ± 1.55	45.21 ± 7.08	18.36 ± 0.40
Single-Player, $C = 0.6$ (Equation (3.6))	23.53 ± 1.52	52.95 ± 8.72	17.25 ± 0.39
Regular UCT, $C = 0.6$ (Equation (3.1))	23.50 ± 1.52	51.85 ± 8.06	17.97 ± 0.40
Adaptive C, $C = 0.6$ (Equation (3.3))	21.53 ± 1.47	55.85 ± 8.85	17.89 ± 0.37
Progressive History (Equation (3.5))	20.07 ± 1.43	46.29 ± 7.41	17.67 ± 0.40
ϵ-Greedy, $\epsilon = 0.6$ (Subsection 3.2.1)	18.90 ± 1.40	34.23 ± 5.32	12.26 ± 0.33
Random(Subsection 3.2.1)	18.20 ± 1.38	35.99 ± 5.64	12.54 ± 0.33
Adaptive ϵ-Greedy (Subsection 3.3.1)	16.97 ± 1.34	34.33 ± 5.42	11.98 ± 0.31

Table 5.4: Evaluation selection strategies in "Open Loop" implementation

5.2.5 Expansion

In Subsection 3.2.2, different strategies to expand the selected node are explained. Table 5.5 shows the results of the different expansion strategies, combined with an "Open Loop" UCT selection ($C = 0.6$) and a random play-out. To evaluate the state during backpropagation, the score heuristic of the GVG-AI Competition (Equation (3.7)) is used, as it performed well in early tests.

Expansion Strategy	**Win Percentage**	**Average Score**	**Average Iterations per Game Cycle**
Random Expansion	23.87 ± 2.16	50.84 ± 10.83	18.42 ± 0.57
Action Based Expansion	22.80 ± 2.12	48.03 ± 10.63	17.74 ± 0.58
All Children Expansion	18.33 ± 1.96	38.49 ± 8.11	10.05 ± 0.38
ϵ-Greedy Expansion	17.87 ± 1.94	-30.08 ± 13.81	10.60 ± 0.42

Table 5.5: Evaluation of expansion strategies

The random expansion performs best of all others in win ratio and number of iterations. The action-based heuristic expands a state based on the performance of an action. With the probability of $\epsilon = 0.4$, a random unexplored child is added to the tree. Otherwise, the most promising unexpanded action is chosen and the corresponding state is added to the tree. Different parameters for ϵ have been tested, as well as using statistics for 1-grams and 3-grams of actions. The results for the different parameters ϵ and N change the performance in the range of confidence bounds. For $N = 1$ and $\epsilon = 0.2$, for example, the win rate is win rate $20.5 \pm 2.0\%$ and for $N = 3$ and $\epsilon = 0.40$, the win rate is $22.6 \pm 2.1\%$. Though additional statistics have to be kept in memory and calculated, the strategy does perform about the same as the simple random expansion.

Adding all unexplored states at once results in a worse performance in all aspects. The main reason for this behavior is that the expansion of the nodes slows down the performance. The ϵ-Greedy expansion performs about the same. The strategy uses the forward model to look one action ahead and take the best state with the probability of $1 - \epsilon = 50\%$. It is disqualified in 5.7% of the games, as it spends too much time. In addition to the lower winning rate, the disqualification leads to a negative score due to a high penalty for being disqualified. Without taking the disqualifications into account, the strategy performs about the same as the All Children Expansion.

The expansion strategy on its own has low impact on the overall performance. Thus, the random expansion is suited for the GVG-AI Competition, due to the limited amount of time.

5.2.6 Play-Out

The play-out is an important step of MCTS, as the result of a play-out has influence on all other steps. Five play-out strategies that are implemented for this thesis, as well as the FAST EVO MCTS PLAYER (GitHub, 2015b) are tested. The FAST EVO MCTS PLAYER is a version of the agent described by Perez *et al.* (2014) and uses the same approach as the (1 + 1) EA developed for this thesis. This version described in the paper of Perez *et al.* (2014) uses the SAMPLE OL-MCTS PLAYER as starting point and combines it with an EA to guide the play-out. The current version that is used for testing adds additional enhancements, but is still in development. Thus, the results are not comparable to the ones mentioned in the original paper. Table 5.6 shows the performance of all the different play-out strategies that have been discussed. All implementations use an "Open Loop" UCT selection and a random expansion strategy. Different configurations of the strategies have been tested and the best ones are presented.

The MAST strategy chooses random actions with the probability of $\epsilon = 0.5$ and uses a decaying factor of 0.1. The NST uses 3-grams of actions and chooses random actions with the probability of $\epsilon = 0.5$ with a decaying factor of 0.0. Thus, the action-based statistics are reset after executing an action. Testing additional decaying factors shows that the MOVE DECAY strategy has no significant impact on the agents performance. The (1 + 1) EA chooses random actions with the probability of $\epsilon = 0.4$. The STGP Evo player has a population size of 100 individuals that are built using the "grow" method and chooses random actions with the probability of $\epsilon = 0.4$. For most other parameters, the STGP Evo player performs well with the ones used by Benbassat and Sipper (2014). The agent uses the same crossover probability of 80%, the same mutation probability of 20%, a $tourSize = 2$ and a $winTourSize = 1$. Only

Game	**MAST**	**NST**	**(1 + 1) EA**	**STGP Evo**	**Random**	**FastEvo**
Aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	98.0 ± 3.9	100.0 ± 0.0	44.0 ± 13.8
Boulderdash	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.0 ± 2.7	0.0 ± 0.0
Butterflies	98.0 ± 3.9	96.0 ± 5.4	94.0 ± 6.6	98.0 ± 3.9	94.0 ± 4.7	88.0 ± 9.0
Chase	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.0 ± 3.9	1.0 ± 2.0	0.0 ± 0.0
Frogs	20.0 ± 11.1	20.0 ± 11.1	6.0 ± 6.6	2.0 ± 3.9	3.0 ± 3.3	0.0 ± 0.0
Missile Command	58.0 ± 13.7	66.0 ± 13.1	54.0 ± 13.8	60.0 ± 13.6	55.0 ± 9.8	32.0 ± 12.9
Portals	10.0 ± 8.3	14.0 ± 9.6	24.0 ± 11.8	14.0 ± 9.6	12.0 ± 6.4	2.0 ± 3.9
Sokoban	14.0 ± 9.6	20.0 ± 11.1	16.0 ± 10.2	10.0 ± 8.3	9.0 ± 5.6	16.0 ± 10.2
Survive Zombies	44.0 ± 13.8	44.0 ± 13.8	36.0 ± 13.3	38.0 ± 13.5	42.0 ± 9.7	26.0 ± 12.2
Zelda	22.0 ± 11.5	16.0 ± 10.2	16.0 ± 10.2	20.0 ± 11.1	10.0 ± 5.9	0.0 ± 0.0
2014 Training Set	**36.6 ± 4.2**	**37.6 ± 4.2**	**34.6 ± 4.2**	**34.2 ± 4.2**	**32.8 ± 2.9**	**21.2 ± 3.6**
Camel Race	8.0 ± 7.5	4.0 ± 5.4	4.0 ± 5.4	4.0 ± 5.4	3.0 ± 3.3	2.0 ± 3.9
Digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Eggomania	6.0 ± 6.6	18.0 ± 10.6	2.0 ± 3.9	4.0 ± 5.4	6.0 ± 4.7	4.0 ± 5.4
Firecaster	2.0 ± 3.9	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.0 ± 2.0	0.0 ± 0.0
Firestorms	8.0 ± 7.5	8.0 ± 7.5	16.0 ± 10.2	4.0 ± 5.4	2.0 ± 2.7	4.0 ± 5.4
Infection	80.0 ± 11.1	78.0 ± 11.5	86.0 ± 9.6	80.0 ± 11.1	78.0 ± 8.1	76.0 ± 11.8
Overload	14.0 ± 9.6	8.0 ± 7.5	32.0 ± 12.9	18.0 ± 10.6	21.0 ± 8.0	0.0 ± 0.0
Pac-Man	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.0 ± 2.7	0.0 ± 0.0
Seaquest	68.0 ± 12.9	60.0 ± 13.6	56.0 ± 13.8	36.0 ± 13.3	40.0 ± 9.6	78.0 ± 11.5
Whac-A-Mole	100.0 ± 0.0	98.0 ± 3.9	96.0 ± 5.4	94.0 ± 6.6	90.0 ± 5.9	76.0 ± 11.8
2014 Validation Set	**28.6 ± 4.0**	**27.4 ± 3.9**	**29.2 ± 4.0**	**24.0 ± 3.7**	**24.3 ± 2.7**	**24.5 ± 3.8**
Bait	6.0 ± 6.6	4.0 ± 5.4	4.0 ± 5.4	12.0 ± 9.0	8.0 ± 5.3	0.0 ± 0.0
Bolo Adventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Brain Man	0.0 ± 0.0	8.0 ± 7.5	0.0 ± 0.0	8.0 ± 7.5	4.0 ± 3.8	6.0 ± 6.6
Chips Challenge	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	10.0 ± 8.3	10.0 ± 5.9	0.0 ± 0.0
Modality	28.0 ± 12.4	26.0 ± 12.2	32.0 ± 12.9	24.0 ± 11.8	24.0 ± 8.4	20.0 ± 11.1
Painter	42.0 ± 13.7	46.0 ± 13.8	58.0 ± 13.7	40.0 ± 13.6	44.0 ± 9.7	44.0 ± 13.8
Real Portals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Real Sokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
The Citadel	14.0 ± 9.6	6.0 ± 6.6	4.0 ± 5.4	10.0 ± 8.3	6.0 ± 4.7	6.7 ± 8.9
Zen Puzzle	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	26.0 ± 12.2	25.0 ± 8.5	6.0 ± 6.6
2015 Training Set	**9.0 ± 2.5**	**9.0 ± 2.5**	**9.8 ± 2.6**	**13.0 ± 2.9**	**12.1 ± 2.0**	**8.7 ± 2.6**
Grand Total	**24.7 ± 2.2**	**24.7 ± 2.2**	**24.5 ± 2.2**	**23.7 ± 2.2**	**23.1 ± 1.5**	**18.3 ± 2.0**

Table 5.6: Win percentages of play-out strategies

the maximum tree depth has to be limited due to the time constraints of the GVG-AI Competition. The initial tree depth of the individual is limited to 5 and the overall maximum depth is limited to 10 instead of 15. Yet, the STGP Evo player can perform 3 fewer iterations per game cycle on average compared to the (1 + 1) EA. The main reason for the slightly decrease in performance is the more complex structure of the individuals in STGP. It takes more time to evaluate, mutate and perform crossover between the individuals compared to the (1 + 1) EA. The original implementation of the STGP Evo player limits the number of assessed actions during each step of the play-out. As the number of actions is limited in the GVG-AI Competition, the custom implementation assesses all actions.

All best performing implementations use a value for $\epsilon \approx 50\%$. However, it has to be mentioned that different values for $\epsilon > 0$ perform without significant differences. For $\epsilon = 0$, thus a greedy play-out, the agents performed worst, especially when combined with Progressive History. Though all agents that used $\epsilon = 0$ performed worst of the NST based agents, the difference is not significant. The FAST EVO MCTS PLAYER at the current state of development does not perform well and even worse than the random player. Still, it shows the impact of the play-out strategy on the overall performance of MCTS. The overall performance of all other play-out strategies is similar. To sum it up, the performance is slightly better than the sample MCTS implementations.

Some games show interesting differences in the agents' performance. NST and MAST, both action-based play-out strategies, seem to perform well in *Frogs*. In other games, the evolutionary approaches that use distance information, especially the (1 + 1) EA, seem to perform better. In *Overload*, for example, the distances to different game objects, such as coins, swords or the exit, are important. Similarly, the distance to the portals in the game *Portals* is important to win the game. In both games, the (1 + 1) EA performs well though still with a similar performance compared to a random play-out. In *Overload*, the (1 + 1) EA performs significantly better than NST.

In *Whack-A-Mole*, an already good performance of the random play-out is increased to perfect play in MAST and almost perfect in NST and the (1 + 1) EA. Overall, the strategies are performing well in the GVG-AI Competition, though there are differences depending on the game. MAST and NST perform similarly. A possible reason could be the parameter k of NST that only incorporates N-grams after they occurred $k = 7$ times. When this threshold is not exceeded, only a 1-gram – thus the same statistic as in MAST – is used. Despite the limited time, 2-grams are frequently used even in slow games, such as *Pac-Man*. 3-grams, though, are less frequently used. The influence of k has not been tested in depth for this thesis. Adding domain knowledge by using distances to opponents in the evolutionary approaches shows no significant improvement. They perform similarly to the statistical, action-based approaches.

5.2.7 Combined Agent

Performance in Different Test Setups

The previous subsections discussed promising techniques to enhance single steps of MCTS. This subsection combines the strategies that performed best or are promising. The first type of combination uses the UCB1-Tuned selection (Equation (3.4)), as it is one of the selection strategies that performed best and does not have parameters to tune. The second type of combinations uses Progressive History, as the same action-based heuristics of NST and MAST – two promising play-out strategies – can be incorporated. All tests are performed with a random expansion and the best heuristic to score the play-out depending on the selection strategy. For tests with the UCB1-Tuned selection, the score heuristic of the GVG-AI Competition is chosen. For tests with Progressive History, the quality-based approach is chosen. Table 5.7 shows the results for agents that use the different play-out strategies. The combinations are tested in the regular setup, as well as a setup with doubled time for choosing an action each game cycle. Each row of the table is based on 6000 games. 3000 games are played in the regular test setup with 40 milliseconds time for executing an action and another 3000 games with 80 milliseconds for executing an action.

It is noticeable that all agents using the UCB1-Tuned selection win fewer games than agents using the same play-out strategy combined with a regular UCT selection. In other words, combining two promising enhancements, such as UCB1-Tuned for selection and NST for play-out, can lower the performance. Therefore, the combination of strategies seems to be important. Using the Progressive history on its own with a random play-out has a low impact on the performance. The strategy even performed significantly worse compared to using the UCB1-Tuned selection as the only enhancement. Using NST or MAST with Progressive History, tough, increases the win percentage significantly from 20.07 ± 1.43 using only

	Win Percentage		**Average Iterations per Game Cycle**		**Average Score**	
Agent	**40 ms**	**80 ms**	**40 ms**	**80 ms**	**40 ms**	**80 ms**
(1 + 1) EA w. UCB1-T.	23.9 ± 1.5	25.3 ± 1.6	$16.5 \pm 0.3\,[1,246]$	$35.2 \pm 0.8\,[1,664]$	47.7 ± 8.0	35.3 ± 9.1
MAST w. UCB1-T.	23.0 ± 1.5	24.9 ± 1.5	$17.3 \pm 0.4\,[1,270]$	$40.6 \pm 0.9\,[1,621]$	49.6 ± 7.0	46.7 ± 8.5
NST w. UCB1-T.	22.6 ± 1.5	24.3 ± 1.5	$16.7 \pm 0.4\,[1,263]$	$40.5 \pm 0.9\,[1,568]$	48.6 ± 7.7	45.1 ± 8.6
STGP Evo w. UCB1-T.	22.2 ± 1.5	25.6 ± 1.6	$12.6 \pm 0.3\,[1,208]$	$29.0 \pm 0.8\,[1,435]$	42.5 ± 6.8	40.9 ± 7.6
sample OL-MCTS	22.8 ± 1.5	24.0 ± 1.5	$21.3 \pm 0.5\,[1,304]$	$47.5 \pm 1.2\,[1,686]$	44.6 ± 7.7	-48.5 ± 12.2
NST w. Prog. Hist.	23.9 ± 1.5	26.1 ± 1.6	$18.0 \pm 0.4\,[1,247]$	$40.1 \pm 1.0\,[1,777]$	43.0 ± 6.9	40.5 ± 8.6
MAST w. Prog. Hist.	24.2 ± 1.5	26.4 ± 1.6	$16.6 \pm 0.4\,[1,324]$	$39.7 \pm 0.9\,[1,760]$	43.3 ± 6.7	40.6 ± 7.9

Table 5.7: Results of combination of different strategies with different time for executing actions

Progressive History as enhancement to 24.2 ± 1.5. Increasing the available time for executing an action slightly improves the results of all agents.

The doubling of available time corresponds to the increase of average iterations per game cycle. As could be expected, the number of iterations and thus play-outs approximately doubles. Still, the number of iterations is low compared to other test setups, where 200 to 400 iterations can be performed in the same amount of time (Pepels *et al.*, 2014b). A similar impact to the iterations, thus the increase in win rate, could be accomplished by using hardware with higher performance.

The average score for the SAMPLE OL-MCTS PLAYER shows a significant decrease. This is due to disqualifications that occurred in consequence of the time management of the player. Most disqualifications occur in *Pac-Man*, *Bolo Adventures* and *Modality*. All of these games are among the games with lowest average iterations per game cycle. *Modality* has a higher number of iterations, though, a higher variance. The time management of the agents takes the average time per iteration into account. A high variance and a low average number of iterations can then lead to a disqualification. The sample agent gets disqualified in 243 of the 3000 games. Therefore, the disqualifications have a significant impact on the average score. Other agents, like the STGP Evo player, get only disqualified eight times. Thus, the disqualification has no significant impact on the score.

Performance in the GVG-AI Competition

The previously mentioned tests mainly focus on single aspects of the agent's performance, such as the percentage of wins. The GVG-AI Competition uses the rules described in Section 2.2 to rank the agents. Furthermore, the tests are performed on different hardware with a different test setup. This section tests the MAST agent and the NST agent on the previously used training set of 2014 and the unknown validation set of 2015. The results of the different training sets can be compared to about 50 agents. After submitting the agent to the test server of the GVG-AI Competition, the agent is tested on each game of the set only once. Thus, the agent only plays 50 games before being ranked. The tests of the previous section have already shown that even running ten times as much games results in a big variance.

Submitting the best performing MAST agent to competition for a first time to play the training set of 2014, the agent won 21 of 50 games and was ranked on 14th place with 26 points. Submitting the agent for a second time, the agent won again 21 games, but was ranked on 25th place with only 12 points. For the validation set of 2015, the agent was both times ranked 11th with 17 of 50 won games and a score of 35 and 36 points. A similar behavior could be observed by submitting the NST agent. The NST agent got worse results for the training set of 2014 and was ranked 22nd and 29th. In the validation set of 2015 the NST agent was ranged 10th and 23rd. At the time of testing, the highest ranked player is the YOLOBOT with 31 of 50 wins and a score of 144 points in the 2015 validation set and 42 of 50 wins and 111 points in the training set of 2014. Thus, the agent has a higher winning rate in both game sets compared to the implementation for this thesis. The SAMPLE OL-MCTS PLAYER was ranked 24th with 23 of 50 wins and 13 points in the training set of 2014. In the validation set of 2015, the sample "Open Loop" implementation performed even worse than the non-"Open Loop" version. The SAMPLE MCTS PLAYER was ranked 20th with 16 of 50 wins and 22 points. In most cases, the MAST agent was ranked higher than the best sample MCTS implementation.

In consequence of the ranking system, the rank is not only dependent of the percentage of wins. In case of the same win ration per game, the score is used as tie breaker. As the number of played games is very low, there are only five different outcomes for the winning rate per game and agent. Thus, the tie breaking rule for the score is used in every game. Due to the high variance in scores, the time is used less as tie breaker. The rule is mainly applied for agents with low win rate and score. As only the first ten places get points according to the rules, the time has low impact on the ranking of an agent.

Compared to the results of the training set of 2014 for this thesis, the agents perform slightly better. The submitted agents won at least 21 of 50 games, thus in average above 42%. Still, it was already mentioned that the results on the test server have a bigger variance. Thus, these results cannot be seen as significantly better, but might indicate an improvement, possibly due to other hardware.

5.3 Evaluation

5.3.1 Challenges

The results presented in the previous section have shown some major challenges for players in the GVG-AI Competition. The main challenge is the short amount of available time. This is even amplified by the rather slow performance of the framework. In *Pac-Man*, for instance, even agents without complex statistics can perform a number of average iterations per game cycle that is close to the number of available actions. Thus, the search tree of MCTS is not deep enough to get reasonable results. As a consequence, complex enhancements show at best small improvements in performance. Some strategies, such as an adaptive ϵ-Greedy selection, performed even worse than a random approach.

An additional challenge is the big random influence while playing games. Most agents introduce some random aspects for tie breaking or exploration. The combination with non-deterministic components of a game leads to a big variance in the wins rate. Especially when evaluating specific game sets or even specific games, this variance gets even higher. The tests show that even running 500 games for each test set, similarly to the GVG-AI Competition, is not enough to get significant results. The test server for the competition even uses only 50 games. To be able to get significant results, more tests have to be performed. As running all three game sets takes more than 19 hours on average, finding good parameters or combinations of strategies takes much time.

In other test setups and domains, such as used by the original implementation of the EvoMCTS player (Benbassat and Sipper, 2014), there is more time available. Additionally, some agents use time in advance to learn models for games. Thus, the strategies have to be tuned to be applicable for the limited time of the GVG-AI Competition. For the EvoMCTS player, for instance, the tree depth for individuals and the population size had to be limited. There are still possibilities to improve the agents, such as a more granular parameter tuning or performance optimization. Still, some techniques, such as UCB1-Tuned, a quality-based play-out evaluation heuristic or NST are promising enhancements. The overall performance of these promising enhancements is similar. However, there are differences depending on the game. In some games, action-based approaches, such as NST or MAST, performed well. In other games, evolutionary approaches performed better. Thus, general game playing is still a major challenge for the MCTS based agent, though the performance improved significantly by using different enhancement compared to a basic MCTS player. The first ranked YOLOBOT seems to perform significantly better. As the source code of all agents will be publicly available, it will be interesting to compare the different approaches.

Another challenge is the overall influence of the framework. Perez *et al.* (2014) used an old version of the framework and created a Knowledge-based Fast-Evo MCTS player that won $49.2 \pm 3.2\%$ of the games in the training set of 2014. The sample MCTS player scored only $21.6 \pm 2.6\%$ at that time. The sample MCTS player at that time did not normalize the scores. After adding a normalization, the performance of the sample player improved. Afterwards the framework as well as the Knowledge-based Fast-Evo MCTS changed. Both changes make it difficult to reproduce results.

5.3.2 Improvements

Despite the challenges mentioned in the previous section, some strategies show promising results. Testing different approaches for scoring a play-out shows the importance of using additional information of a game state. Starting with the basic MCTS heuristic that only uses information about wins and losses, the agent won $19.6 \pm 2.0\%$ of the played games. The agent's performance can be increased significantly by incorporating additional information, such as the pay-out depth or the score. The handcrafted heuristic used by the GVG-AI Competition's MCTS players, as well as the quality-based approach by Pepels *et al.* (2014a) perform well in the GVG-AI Competition. Though the quality-based approach is a bit more complex, it has the advantage of being able to handle a bigger variety of scores. The heuristic used by the GVG-AI Competition adds fixed high bonus to the score in case of a win. If a score is higher than this bonus or close to it, the focus would be less on winning games. Still, both heuristics performed well and improved the results up to a win rate of $25.9 \pm 2.2\%$ by using a version of the quality-based approach with tuned parameters. Though a higher score has not necessarily to result in a higher win ratio (Subsection 5.2.3), the score has big influence on the ranking in GVG-AI Competition (Subsection 5.2.7).

Additional improvements can be observed for different selection strategies. Early tests show that storing the game state inside the nodes of MCTS does not perform well due to the non-deterministic character of the games. The so-called "Open Loop" implementation slightly improves the results and significantly improves the number of iterations per game cycle. The SAMPLE MCTS PLAYER uses a regular UCT selection with $C = 1.4$. A custom implementation with a tuned parameter $C = 0.6$ performed only insignificantly better. The best results, though only insignificantly better, are obtained by using the UCB1-Tuned selection. Still, the selection strategy has the advantage to have no tunable parameter C, which can be helpful in general game playing.

For expansion, random or action-based strategies performed similarly well. All other strategies that are tested in this thesis decreased the performance.

Despite the low number of iterations per game cycle, different play-out strategies with different grades of complexity are applicable for the GVG-AI Competition. However, action-based strategies – such as NST or MAST – as well as evolutionary approaches, show no significant impact on the overall performance. Nevertheless, in some specific games the different strategies show a significant improvement in the win ratio. Despite the higher complexity of play-out strategies like NST, they perform a similar number of iterations per game cycle compared to a simple random play-out (Table 5.2).

Tuning of different parameters changed the behavior of different techniques. Even in the GVG-AI Competition that focuses on general game playing, some domain-specific aspects have to be taken into account. Aspects, such as the time limitation, have influence on the tuning of parameters for enhancements. Taking these aspects into account, the parameters of even complex enhancements can be tuned to perform in the GVG-AI Competition. The STGP, for instance, can be used, even though the performance is not significantly better compared to other techniques. One of the enhancements that performs best in the different test setups is the MAST player that uses Progressive History. Compared to the SAMPLE OL-MCTS PLAYER that was used as starting point, the MAST player performs better in all used test environments: the test server for the GVG-AI Competition, as well as the custom test setup. Considering that the SAMPLE MCTS PLAYER was ranked third during the competition in 2014 and that the SAMPLE OL-MCTS PLAYER performs better, this is a good result. Furthermore, the results of this thesis can be used as basis to start additional enhancements. Some of them are proposed in the next chapter.

Chapter 6

Conclusion

Different strategies and heuristics to tune MCTS are described and evaluated in this thesis in the context of the GVG-AI Competition. During the evaluation, some techniques improved the performance and others showed some challenges. This chapter summarizes these aspects by answering the research question and the problem statement and gives an outlook for future research.

Chapter contents: Research Questions, Problem Statement & Future Research

6.1 Research Questions

In Section 1.3, research questions are stated to guide the research for this thesis. The results provided in Section 5.2 help to give an answer to these questions.

RQ 1: Which existing approaches can be used to improve MCTS without using domain knowledge?

This thesis introduces existing approaches to improve the performance of MCTS in the GVG-AI Competition. These enhancements can be applied to the steps of MCTS as well as the overall scoring of play-outs. Some of the enhancements have to be adapted to the restrictions of the GVG-AI Competition, such as the limited amount of time to execute actions. Promising and well performing combinations of parameters have been discussed as well as promising techniques without any parameter. The score heuristic for play-outs is identified as an important component for improvement. Incorporating the score with a quality-based heuristic or a hand crafted heuristic that is used by the GVG-AI Competition improved the performance significantly. For selection, UCT with different adaptations performed similarly well. However, UCB1-Tuned has the advantage not to have parameters that have to be tuned and seems to perform best. For expansion, a simple random strategy seems to perform best with similar results as an action-based approach. The performance of the random strategy is even significantly better than other, more complex approaches. During play-out, random, statistical and evolutionary approaches perform similarly. Some of the strategies use a kind of domain knowledge, such as distances between game objects or a score obtained by a forward model. Using distances has only insignificant impact on the overall performance, though it changes the behavior of the agent in some games. The score as another domain-specific aspect helps to improve agent's overall performance significantly. Overall, the approaches use no domain knowledge or domain knowledge that can be used for many different games.

RQ 2: Which combination of techniques can be used to improve the performance of MCTS best in general video games?

The tests have shown promising combinations of enhancements for MCTS for the GVG-AI Competition. The combination of MAST and NST with Progressive History perform well in all used test setups. Nevertheless, simple agents with random play-out strategies, UCT selection with a tuned parameter C and different heuristics for play-out evaluation perform insignificantly better. Especially the heuristic to score a play-out is of importance, as – due to the restrictions of the GVG-AI Competition – terminal game states are rare during simulation.

RQ 3: What are the strengths and weaknesses of the different approaches for general video games and how can they be enhanced?

Different types of agents have shown strength in different games. Action-based agents, such as MAST, perform better in some games compared to evolutionary approaches, like the (1 + 1) EA, and vice versa. Still, the overall performance is similar regarding the win rate and the score. Furthermore, the agents perform fewer iterations per game cycle compared to simpler approaches. The number of performed iterations can be improved by additional performance optimization. A possible solution to incorporate both strength of action-based and evolutionary approach could be the STGP Evo player. The approach is designed to combine different features with different types and learn well performing combinations. Thus, additional nodes and combination of nodes for the individual could incorporate both strengths. This could be a possible way to incorporate domain-specific knowledge, such as distances to game objects, as well as domain-independent knowledge.

RQ 4: How well does the agent perform under different constraints?

The major constraint of the GVG-AI Competition is the time limit for executing an action. Doubling this time slightly improves the performance of all agents. However, even a doubled time allows the agents to perform fewer play-outs compared to other real time frameworks. Despite the limited time, testing an agent takes several hours per game set to reduce the variance in performance. The number of played games has even influence on the ranking used for the GVG-AI Competition because of tie breaking rules. Thus, the test setup is important. The developed agents for this thesis performed well in all test setups, though there are still possibilities for improvement.

6.2 Problem Statement

All research questions have been answered at least partially. These questions narrow down the overall goal for this thesis to the following problem statement.

Problem Statement: *Which techniques for enhancing an MCTS based agent improves its performance in the GVG-AI Competition?*

These techniques are simple ones, such as random strategies, as well as more complex ones that use, for instance, Evolutionary Algorithms. The results have shown that the performance is significantly improved compared to a regular MCTS agent by applying these techniques. The best performing enhancements incorporate additional statistics to different steps of MCTS. Using the score of a game state significantly improved the results. Different promising approaches for these score heuristics have been discussed. Other techniques that use action-based scores or Evolutionary Algorithms only improved the performance insignificantly. However, the developed agent performs better than the third ranked SAMPLE MCTS PLAYER of last year's GVG-AI Competition in most test setups.

Testing the agent with a validation set of games, the agent is ranked at place 10 out of 50, though the variance of the test setup is high. Despite the high variance, the top-ranked participant of the competition seems to perform significantly better. Overall, the agent developed for this thesis performs well in the GVG-AI Competition, though there are still points for improvement. The agent and all of the strategies will be publicly available to give the opportunity for future research.

6.3 Future Research

Several points of improvement and future research are stated in this thesis. Not all combinations of the presented strategies have been tested. Furthermore, the combinations have to be adjusted towards each other. A heuristic that scores a play-out in MCTS has, for example, influence on the selection that uses the score. Thus, a change in the score heuristic can have influence on parameters of the selection. Some, but not all of these aspects, have been tested.

Using the score of a game state provided by the competition's forward model improved the agent's performance significantly. Additional information could be incorporated. The forward model, for example, provides a list of events that happened during the game. In case of a terminal game state, the last event, such as a collision between the avatar and a NPC, could reveal additional information. The distance as another example is already incorporated in the evolutionary play-out strategies presented in this thesis. Similarly, events could be added as new features. In case of the STGP Evo player, new nodes that use these events could be added. In addition to new nodes, the combination of nodes could be tested.

As the time for executing an action is limited by the GVG-AI Competition, additional enhancements that optimize the behavior of the agent could be interesting. The STGP Evo player can build individuals where some sub-trees have no influence on the fitness of the individual. Removing these so-called introns can improve the performance of the algorithm (Benbassat and Sipper, 2014). In addition, there are more general approaches to improve the performance of all play-out strategies. Finnsson (2012) uses the change in score during the play-out to influence the play-out depth. Using the stability of the score, promising play-outs can be evaluated in more depth and non-promising play-outs can be cut off.

After the GVG-AI Competition is over, the source code will be publicly available (cf. Perez, 2015) under the player name MaastCTS. Thus, the different strategies can be used, improved or simply used for benchmarking. Furthermore, the agent can be compared to other submitted agents, for instance, the YOLOBOT that seems to perform well.

References

Arifovic, Jasmina (1994). Genetic algorithm learning and the cobweb model. *Journal of Economic Dynamics and Control*, Vol. 18, No. 1, pp. 3–28. [25]

Auer, Peter, Cesa-Bianchi, Nicolò, and Fischer, Paul (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. ISSN 0885–6125. [17, 18]

Benbassat, Amit and Sipper, Moshe (2010). Evolving Lose-Checkers Players using Genetic Programming. *2010 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 30–37, IEEE. [26]

Benbassat, Amit and Sipper, Moshe (2014). EvoMCTS: A Scalable Approach for General Game Learning. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 382–394. [1, 2, 3, 18, 26, 33, 34, 35, 42, 44, 48, 53]

Browne, Cameron B., Powley, Edward, Whitehouse, Daniel, Lucas, Simon M., Cowling, Peter I., Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43. ISSN 1943–068X. [13, 14]

Chaslot, Guillaume M. J-B., Winands, Mark H. M., Herik, H. Jaap van den, Uiterwijk, Jos W. H. M., and Bouzy, Bruno (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [14, 15, 16, 19]

Clark, Christopher and Storkey, Amos (2014). Teaching Deep Convolutional Neural Networks to Play Go. *arXiv preprint arXiv:1412.3409*. [1]

Corcoran, Arthur L. and Sen, Sandip (1994). Using Real-Valued Genetic Algorithms to Evolve Rule Sets for Classification. *Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence*, Vol. 1, pp. 120–124. [25]

Coulom, Rémi (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* (eds. H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83. Springer Berlin Heidelberg. ISBN 978–3–540–75537–1. [2, 13]

Decoster, Bertrand (2014). GGP Competition 2014. `http://games.stanford.edu/index.php/ggp-competition`. [1, 5, 6]

Droste, Stefan, Jansen, Thomas, and Wegener, Ingo (2002). On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, Vol. 276, No. 1, pp. 51–81. [26, 29, 30]

Ebner, Marc, Levine, John, Lucas, Simon M., Schaul, Tom, Thompson, Tommy, and Togelius, Julian (2013). Towards a Video Game Description Language. *Artificial and Computational Intelligence in Games* (eds. Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 85–100. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany. ISBN 978–3–939897–62–0, ISSN 1868–8977. [7]

Falcone, Marco A., Lopes, Heitor S., and dos Santos Coelho, Leandro (2008). Supply chain optimisation using Evolutionary Algorithms. *International Journal of Computer Applications in Technology*, Vol. 31, No. 3, pp. 158–167. [25]

Ferrucci, David A., Levas, Anthony, Bagchi, Sugato, Gondek, David, and Mueller, Erik T. (2013). Watson: Beyond Jeopardy! *Artificial Intelligence*, Vol. 199–200, pp. 93–105. [1]

Finnsson, Hilmar (2007). CADIA-Player: A General Game Playing Agent. M.Sc. thesis, Reykjavík University, Iceland. [16]

Finnsson, Hilmar (2012). *Simulation-Based General Game Playing*. Ph.D. dissertation, Reykjavík University, Iceland. [53]

Finnsson, Hilmar and Björnsson, Yngvi (2008). Simulation-Based Approach to General Game Playing. *AAAI*, Vol. 8, pp. 259–264. [3, 13, 16]

Formula One World Championship Limited (2015). Formula 1 - The Official F1 Website - Rules And Regulations - Points. `http://www.formula1.com/inside_f1/rules_and_regulations/sporting_regulations/8681/`. [6]

Freitas, Alex A. (2003). A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery. *Advances in evolutionary computing* (eds. Ashish Ghosh and Shigeyoshi Tsutsui), Natural Computing Series, pp. 819–845. Springer Berlin Heidelberg. ISBN 978–3–642–62386–8. [25]

GECCO (2015). The Genetic and Evolutionary Computation Conference - 2015. `http://www.sigevo.org/gecco-2015/`. [6]

GitHub (2015a). The framework for the General Video Game Competition 2014. `https://github.com/EssexUniversityMCTS/gvgai`. [37]

GitHub (2015b). EvoMCTS - Code to work with Fast Evo MCTS. `https://github.com/diegopliebana/EvoMCTS/`. [44]

Han, Jiawei, Kamber, Micheline, and Pei, Jian (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann, third edition. ISBN 978–0–12–381479–1. [2]

Harvey, Inman (2011). The Microbial Genetic Algorithm. *Advances in Artificial Life. Darwin Meets von Neumann* (eds. George Kampis, István Karsai, and Eörs Szathmáry), Vol. 5778 of *Lecture Notes in Computer Science*, pp. 126–133. Springer Berlin Heidelberg. ISBN 978–3–642–21313–7. [10]

Hsu, Feng-Hsiung (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press. [1]

IEEE CIG (2015). 2015 IEEE Conference on Computational Intelligence and Games. `http://cig2015.nctu.edu.tw/`. [6]

Knuth, Donald E. and Moore, Ronald W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [2]

Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer Berlin Heidelberg. ISBN 978–3–540–45375–8. [2, 14]

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Vol. 1. MIT Press. ISBN 0–262–11170–5. [25, 27]

Lazarus, Christopher and Hu, Huosheng (2001). Using Genetic Programming to Evolve Robot Behaviours. *Proceedings of the 3rd British Conference on Autonomous Mobile Robotics and Autonomous Systems*, Manchester. [25]

Levine, John, Congdon, Clare Bates, Ebner, Marc, Kendall, Graham, Lucas, Simon M., Miikkulainen, Risto, Schaul, Tom, and Thompson, Tommy (2013). General Video Game Playing. *Artificial and Computational Intelligence in Games* (eds. Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 77–83. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany. ISBN 978–3–939897–62–0, ISSN 1868–8977. [1, 3, 5]

Li, Yun, Ng, Kim Chwee, Murray-Smith, David J., Gray, Gary J., and Sharman, Ken C. (1996). Genetic algorithm automated approach to the design of sliding mode control systems. *International Journal of Control*, Vol. 63, No. 4, pp. 721–739. [25]

Lucas, Simon M., Samothrakis, Spyridon, and Perez, Diego (2014). Fast Evolutionary Adaptation for Monte Carlo Tree Search. *Applications of Evolutionary Computation* (eds. Anna I. Esparcia-Alcázar and Antonio M. Mora), Vol. 8602 of *Lecture Notes in Computer Science*, pp. 349–360. Springer Berlin Heidelberg. ISBN 978–3–662–45522–7. [29, 30]

Luke, Sean, Panait, Liviu, Balan, Gabriel, Paus, Sean, Skolicki, Zbigniew, Kicinger, Rafal, Popovici, Elena, Sullivan, Keith, Harrison, Joseph, Bassett, Jeff, Hubley, Robert, Desai, Ankur, Chircop, Alexander, Compton, Jack, Haddon, William, Donnelly, Stephen, Jamil, Beenish, Zelibor, Joseph, Kangas, Eric, Abidi, Faisal, Mooers, Houston, O'Beirne, James, Talukder, Khaled A., and McDermott, James (2015). ECJ 22 - A Java-based Evolutionary Computation Research System. `https://cs.gmu.edu/~eclab/projects/ecj/`. [29]

Montana, David J. (1995). Strongly Typed Genetic Programming. *Evolutionary Computation*, Vol. 3, No. 2, pp. 199–230. [2, 26, 27, 28]

Nijssen, J. (Pim) A. M. and Winands, Mark H. M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games* (eds. H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat), Vol. 6515 of *Lecture Notes in Computer Science*, pp. 238–249. Springer Berlin Heidelberg. ISBN 978–3–642–17927–3. [17]

Pepels, Tom, Tak, Mandy J. W., Lanctot, Marc, and Winands, Mark H. M. (2014a). Quality-based Rewards for Monte-Carlo Tree Search Simulations. *21st European Conference on Artificial Intelligence (ECAI 2014)* (eds. Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan), Vol. 263 of *Frontiers in Artificial Intelligence and Applications*, pp. 705–710, IOS Press. [19, 20, 41, 48]

Pepels, Tom, Winands, Mark H. M., and Lanctot, Marc (2014b). Real-Time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 3, pp. 245–257. [39, 47]

Perez, Diego (2015). The General Video Game AI Competition - 2014. `http://www.gvgai.net/`. [5, 6, 10, 53]

Perez, Diego, Samothrakis, Spyridon, and Lucas, Simon M. (2014). Knowledge-based Fast Evolutionary MCTS for General Video Game Playing. *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 68–75, IEEE. [2, 18, 25, 26, 29, 30, 33, 44, 48]

Perez, Diego, Samothrakis, Spyridon, Togelius, Julian, Schaul, Tom, Lucas, Simon M., Couëtoux, Adrien, Lee, Jeyull, Lim, Chong-U, and Thompson, Tommy (2015). The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*. ISSN 1943–068X. `http://dx.doi.org/10.1109/TCIAIG.2015.2402393`. [2, 6, 10, 11, 21, 37, 39]

Powley, Edward J., Cowling, Peter I., and Whitehouse, Daniel (2014). Information capture and reuse strategies in Monte Carlo Tree Search, with applications to games of hidden information. *Artificial Intelligence*, Vol. 217, pp. 92–116. [13]

Robles, David and Lucas, Simon M. (2009). A Simple Tree Search Method for Playing Ms. Pac-Man. *2009 IEEE Symposium on Computational Intelligence and Games (CIG)*, pp. 249–255, IEEE. [1]

Schadd, Maarten P .D., Winands, Mark H. M., Herik, H. Jaap van den, Chaslot, Guillaume M. J-B., and Uiterwijk, Jos W. H. M. (2008). Single-Player Monte-Carlo Tree Search. *Computers and Games* (eds. H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 1–12. Springer Berlin Heidelberg. ISBN 978–3–540–87607–6. [17]

Schaul, Tom (2014). An Extensible Description Language for Video Games. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 325–331. [7]

Schepers, Colin (2012). Automatic Decomposition of Continuous Action and State Spaces in Simulation-Based Planning. M.Sc. thesis, Maastricht University, the Netherlands. [17]

Sutton, Richard S. and Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press. [15]

Tak, Mandy J. W., Winands, Mark H. M., and Björnsson, Yngvi (2012). N-Grams and the Last-Good-Reply Policy applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 73–83. [16, 18]

Tak, Mandy J. W., Winands, Mark H. M., and Björnsson, Yngvi (2014). Decaying Simulation Strategies. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 395–406. ISSN 1943–068X. [18]

The Robocup Federation (2015). RoboCup. `http://www.robocup.org/`. [6]

Appendix A

UML

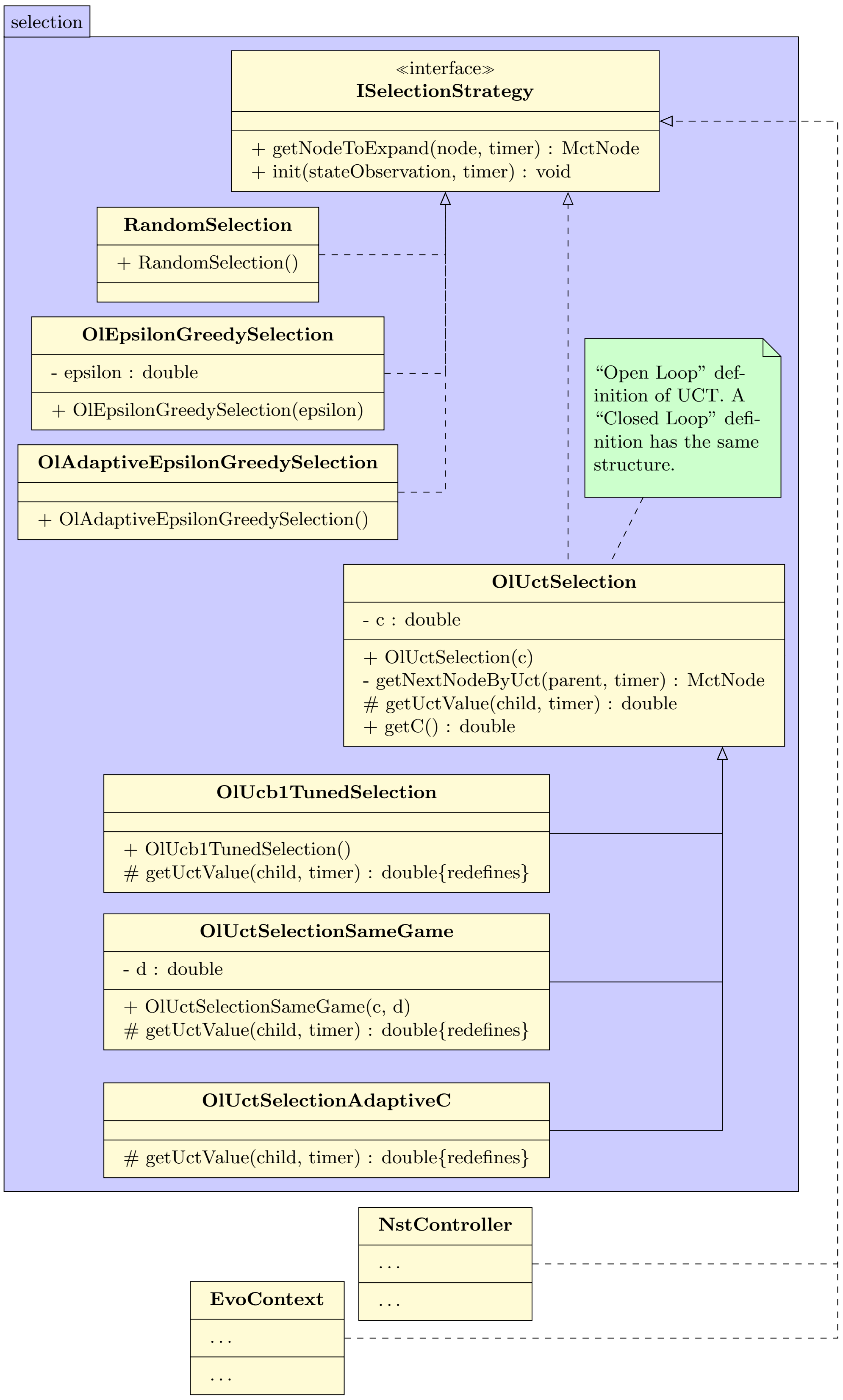

selection
≪interface≫
ISelectionStrategy
+ getNodeToExpand(node, timer) : MctNode
+ init(stateObservation, timer) : void
RandomSelection
+ RandomSelection()
OlEpsilonGreedySelection
- epsilon : double
+ OlEpsilonGreedySelection(epsilon)
OlAdaptiveEpsilonGreedySelection
+ OlAdaptiveEpsilonGreedySelection()
“Open Loop” definition of UCT. A “Closed Loop” definition has the same structure.
OlUctSelection
- c : double
+ OlUctSelection(c)
- getNextNodeByUct(parent, timer) : MctNode
# getUctValue(child, timer) : double
+ getC() : double
OlUcb1TunedSelection
+ OlUcb1TunedSelection()
# getUctValue(child, timer) : double{redefines}
OlUctSelectionSameGame
- d : double
+ OlUctSelectionSameGame(c, d)
# getUctValue(child, timer) : double{redefines}
OlUctSelectionAdaptiveC
# getUctValue(child, timer) : double{redefines}
NstController
. . .
. . .
EvoContext
. . .
. . .

Figure A.1: Selection strategies for MCTS

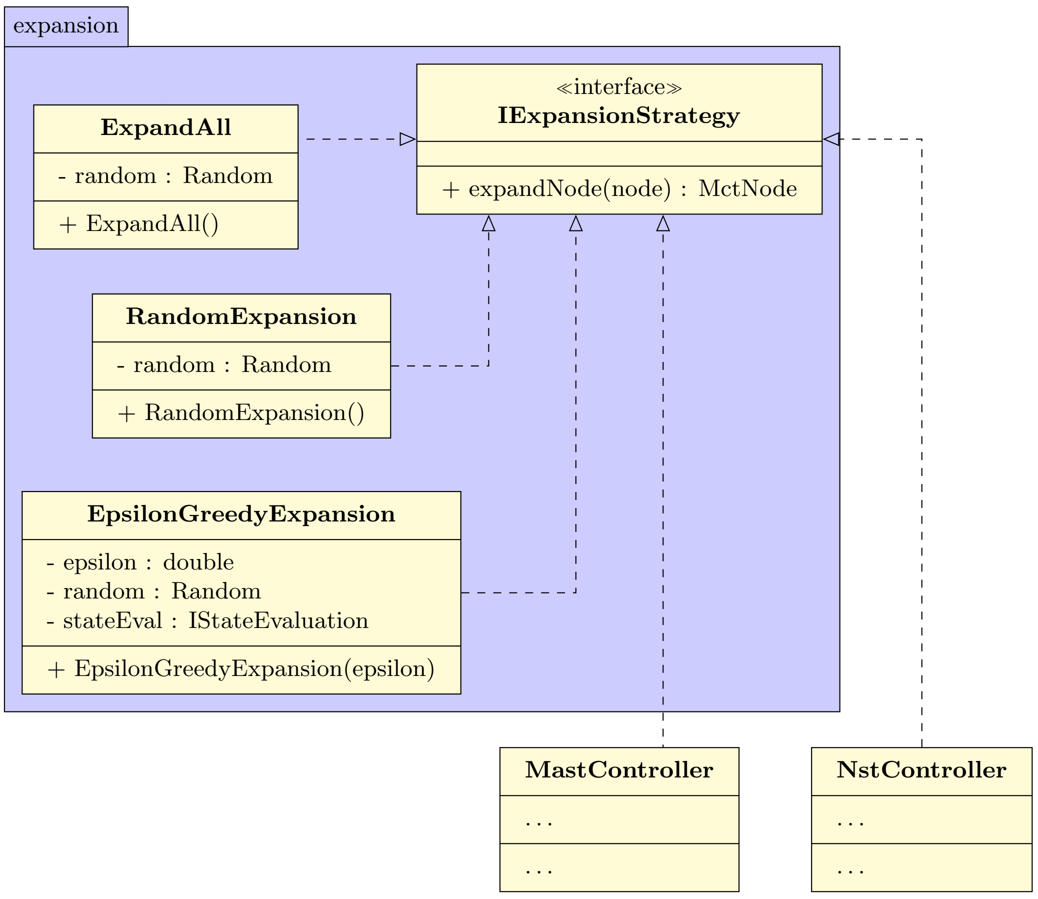


Figure A.2: Expansion strategies for MCTS

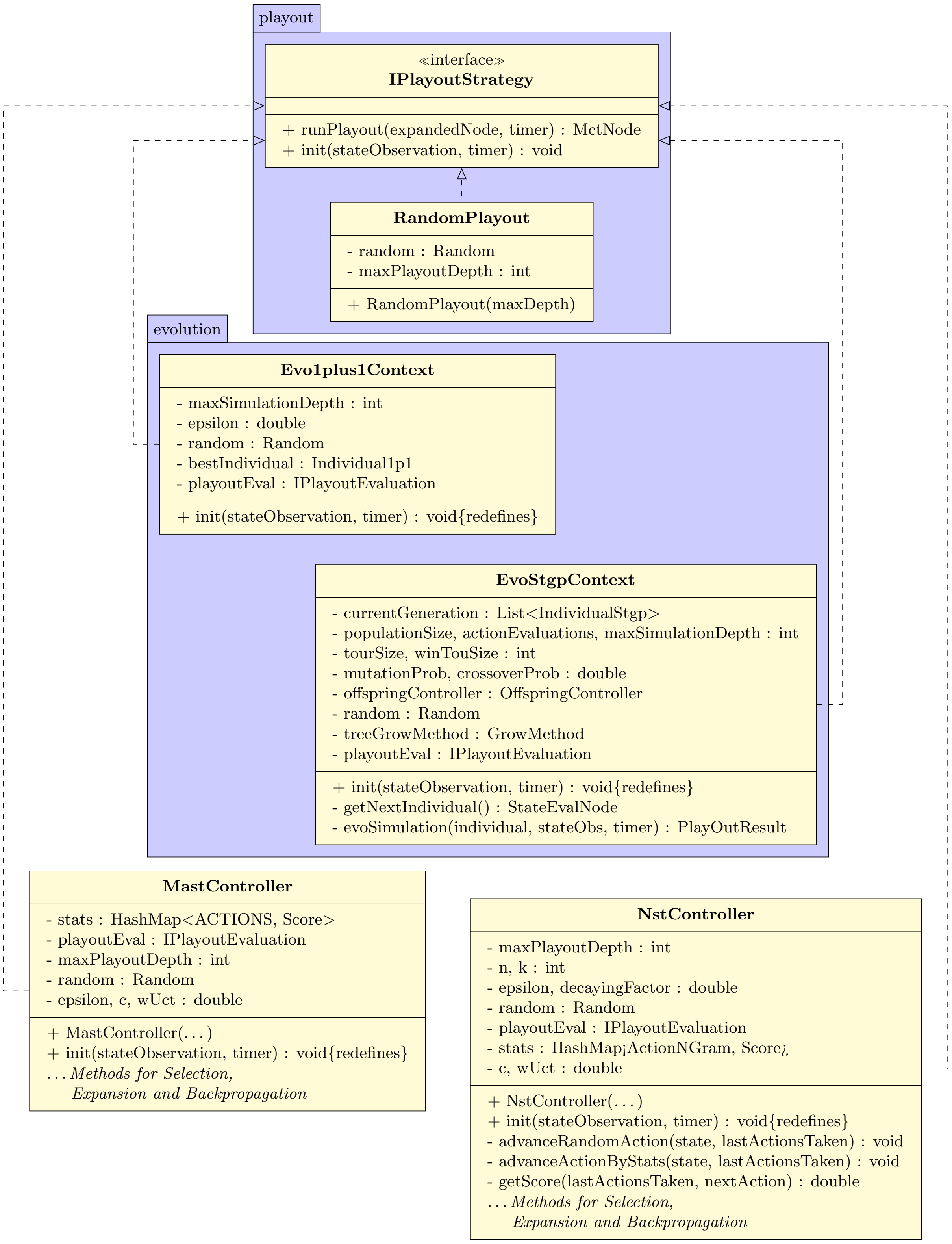

playout
≪interface≫
IPlayoutStrategy
+ runPlayout(expandedNode, timer) : MctNode
+ init(stateObservation, timer) : void
RandomPlayout
- random : Random
- maxPlayoutDepth : int
+ RandomPlayout(maxDepth)
evolution
Evo1plus1Context
- maxSimulationDepth : int
- epsilon : double
- random : Random
- bestIndividual : Individual1p1
- playoutEval : IPlayoutEvaluation
+ init(stateObservation, timer) : void{redefines}
EvoStgpContext
- currentGeneration : List<IndividualStgp>
- populationSize, actionEvaluations, maxSimulationDepth : int
- tourSize, winTouSize : int
- mutationProb, crossoverProb : double
- offspringController : OffspringController
- random : Random
- treeGrowMethod : GrowMethod
- playoutEval : IPlayoutEvaluation
+ init(stateObservation, timer) : void{redefines}
- getNextIndividual() : StateEvalNode
- evoSimulation(individual, stateObs, timer) : PlayOutResult
MastController
- stats : HashMap<ACTIONS, Score>
- playoutEval : IPlayoutEvaluation
- maxPlayoutDepth : int
- random : Random
- epsilon, c, wUct : double
+ MastController(. . .)
+ init(stateObservation, timer) : void{redefines}
. . . Methods for Selection,
Expansion and Backpropagation
NstController
- maxPlayoutDepth : int
- n, k : int
- epsilon, decayingFactor : double
- random : Random
- playoutEval : IPlayoutEvaluation
- stats : HashMap¡ActionNGram, Score¿
- c, wUct : double
+ NstController(. . .)
+ init(stateObservation, timer) : void{redefines}
- advanceRandomAction(state, lastActionsTaken) : void
- advanceActionByStats(state, lastActionsTaken) : void
- getScore(lastActionsTaken, nextAction) : double
. . . Methods for Selection,
Expansion and Backpropagation

Figure A.3: Play-out strategies for MCTS

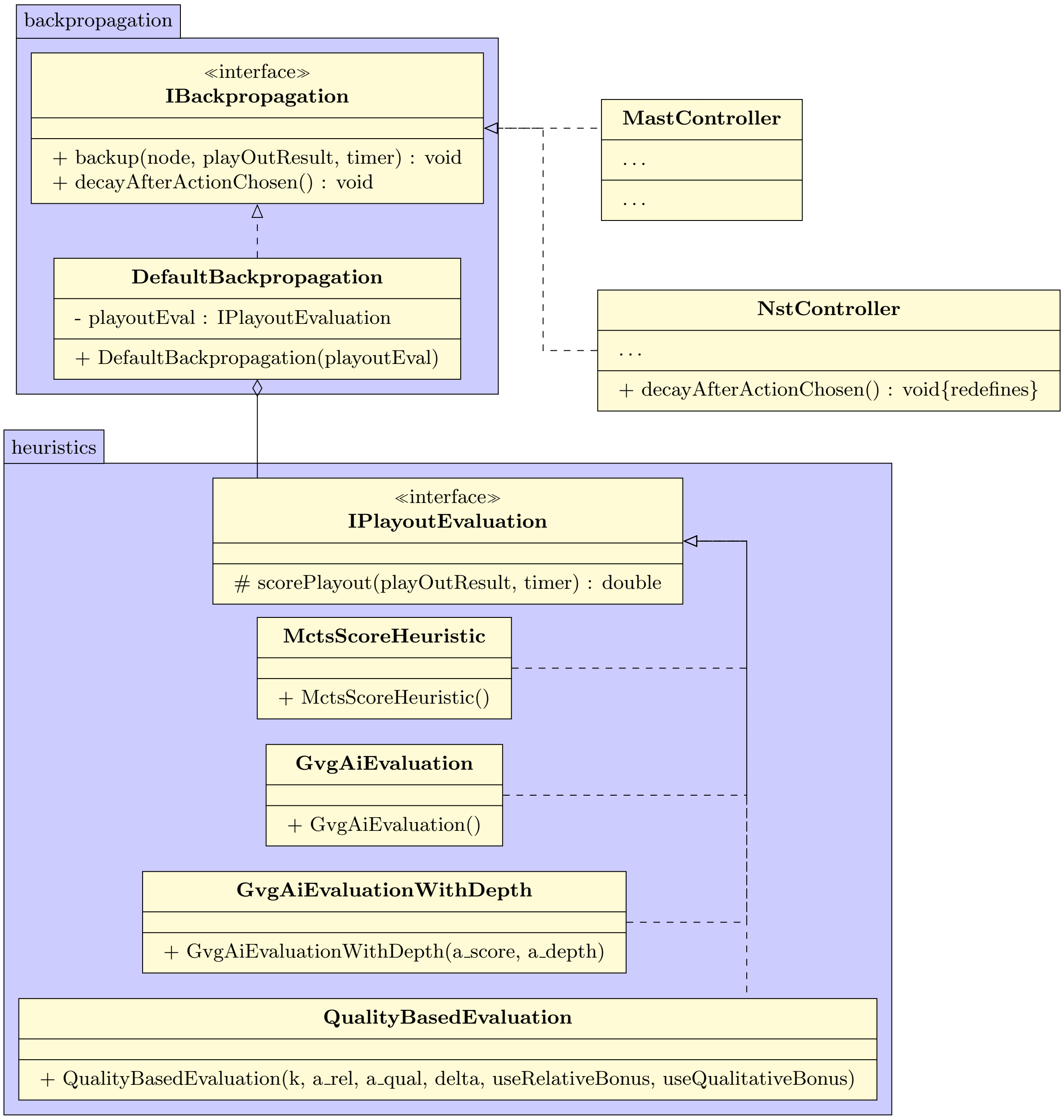

backpropagation
«interface»
IBackpropagation
+ backup(node, playOutResult, timer) : void
+ decayAfterActionChosen() : void
DefaultBackpropagation
- playoutEval : IPlayoutEvaluation
+ DefaultBackpropagation(playoutEval)
MastController
. . .
. . .
NstController
. . .
+ decayAfterActionChosen() : void{redefines}
heuristics
«interface»
IPlayoutEvaluation
# scorePlayout(playOutResult, timer) : double
MctsScoreHeuristic
+ MctsScoreHeuristic()
GvgAiEvaluation
+ GvgAiEvaluation()
GvgAiEvaluationWithDepth
+ GvgAiEvaluationWithDepth(a_score, a_depth)
QualityBasedEvaluation
+ QualityBasedEvaluation(k, a_rel, a_qual, delta, useRelativeBonus, useQualitativeBonus)

Figure A.4: Backpropagation strategies for MCTS

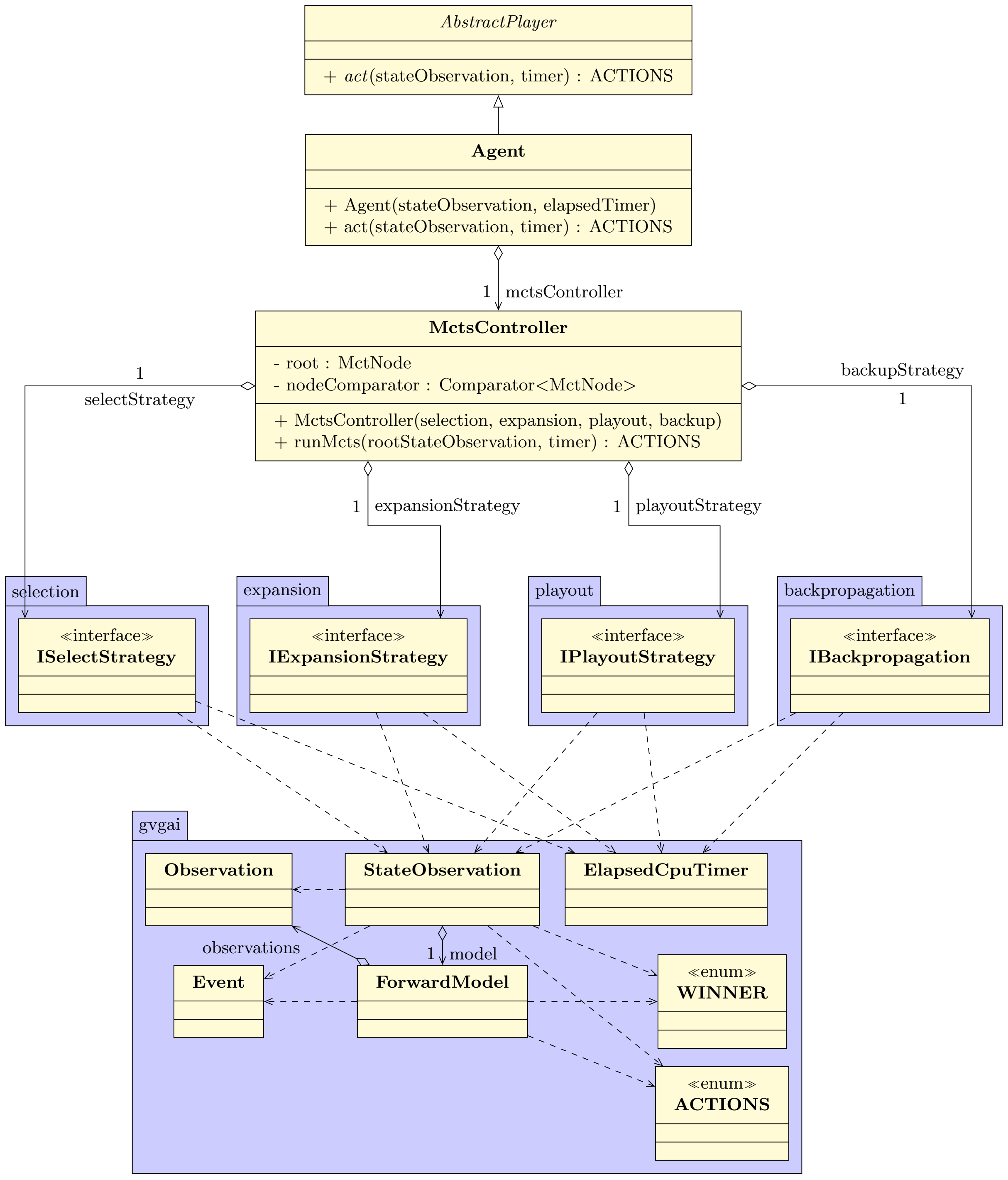


Figure A.5: Architecture of MCTS based agent with important classes of GVG-AI Competition