Master Thesis

Enhancements for Real-Time Monte-Carlo Tree Search in General Video Game Playing

Dennis J. N. J. Soemers

Master Thesis DKE 16-11

Thesis submitted in partial fulfillment
of the requirements for the degree of Master of Science
of Artificial Intelligence at the Department of
Data Science and Knowledge Engineering
of the Maastricht University

Thesis Committee:

Dr. Mark H. M. Winands
Chiara F. Sironi, M.Sc.
Dr. ir. Kurt Driessens

Maastricht University
Faculty of Humanities and Sciences
Department of Data Science and Knowledge Engineering
Master Artificial Intelligence

July 1, 2016

Preface

This master thesis is written at the Department of Data Science and Knowledge Engineering, Maastricht University. The thesis describes my research into enhancing an agent using Monte-Carlo Tree Search for the real-time domain of General Video Game Playing. Parts of this research have been accepted for publication in the conference proceedings of the 2016 IEEE Conference on Computational Intelligence and Games. First, I would like to thank Dr. Mark Winands, not only for dedicating much time and effort supervising this thesis, but also my bachelor thesis and research internship in previous years. Additionally, I would like to thank Chiara Sironi, M.Sc., for also supervising this thesis. I would also like to thank my friend Taghi Aliyev for frequently helping me out. Finally, I would like to thank my family for supporting me.

Dennis Soemers
Maastricht, June 2016

Abstract

General Video Game Playing (GVGP) is a field of Artificial Intelligence where agents play a variety of real-time video games that are unknown in advance. This is challenging because it requires fast decision-making (40 milliseconds per decision), and it is difficult to use game-specific knowledge. Agents developed for GVGP are evaluated every year in the GVGP Competition.

Monte-Carlo Tree Search (MCTS) is a search technique for game playing that does not rely on domain-specific knowledge. MCTS is known to perform well in related domains, such as General Game Playing. In 2015, MCTS-based agents were outperformed by other techniques in GVGP. The Iterated Width (IW) algorithm, which originates from classic planning, performed particularly well.

In this thesis, it is investigated how MCTS can be enhanced to raise its performance to a competitive level in GVGP. This is done by evaluating enhancements known from previous research, extending some of them, and introducing new enhancements. These are partially inspired by IW. The investigated enhancements are *Breadth-First Tree Initialization and Safety Prepruning*, *Loss Avoidance*, *Novelty-Based Pruning*, *Progressive History*, *N-Gram Selection Technique*, *Tree Reuse*, *Knowledge-Based Evaluations*, *Deterministic Game Detection*, and *Temporal-Difference Tree Search*.

The effect that these enhancements have on the performance of MCTS in GVGP is experimentally evaluated using sixty different GVGP games of the GVG-AI framework. Most of them are shown to provide statistically significant increases in the average win percentage individually. Among the agents combining multiple enhancements, the agent with the best win percentage (48.4%) performs significantly better than the baseline MCTS implementation of this thesis (31.0%), and close to the level of YBCriber, which was the winning agent of the GVGP competition at the IEEE CEEC 2015 conference (52.4%).

Contents

Chapter 1

Introduction

This chapter provides an introduction to this thesis. It discusses Artificial Intelligence for games, and the concept of General Video Game Playing, which is the focus of this thesis. The problem statement and four research questions are described next. Finally, this chapter contains an outline of the remainder of the thesis.

1.1 Artificial Intelligence and Games

One of the main topics of research in Artificial Intelligence (AI) is game-playing. In many games, it is a challenging problem to find good decisions to make. The $\alpha\beta$ search algorithm (Knuth and Moore, 1975) and a number of enhancements have eventually led to the Deep Blue system (Campbell, Hoane Jr, and Hsu, 2002) defeating the human world chess champion Garry Kasparov in 1997. In the game of Go, agents based on the $\alpha\beta$ technique have not been able to compete with expert human players. The Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007) algorithm has increased the performance of AI in this domain, and has been used by the AlphaGo (Silver *et al.*, 2016) program, which beat the 9-dan professional human player Lee Sedol in 2016. MCTS also has applications in domains other than game-playing (Browne *et al.*, 2012), which indicates that research in game-playing algorithms can also be useful for "real-world" problems.

Even though the basic techniques used by game-playing agents such as Deep Blue and AlphaGo are applicable in a variety of domains, they also rely on domain-specific knowledge to be effective (for instance in the form of heuristics, or offline training using domain-specific data). This means that these agents are not able to play other games than the ones they were specifically programmed to play. To promote research in more generally applicable techniques, the General Game Playing (GGP) competition (Genesereth, Love, and Pell, 2005) is organized annually. In GGP, agents should be able to play any game of which the rules are specified in a Game Description Language (GDL). The focus in GGP is placed on abstract games. General Video Game Playing (GVGP) (Levine *et al.*, 2013) is a similar concept, but it focuses on real-time video games instead of abstract games. The Arcade Learning Environment (Bellemare *et al.*, 2013) is a similar framework that can be used to develop agents that play games of the Atari 2600 game console. This thesis focuses on GVGP.

1.2 General Video Game Playing (GVGP)

To test the performance of different techniques for GVGP, the GVGP Competition is organized annually. The first GVGP Competition was held at the IEEE Conference on Computational Intelligence and Games (CIG) in 2014 (Perez *et al.*, 2016). Three sets of ten different real-time video games per set, for a total of thirty games, were used in this competition. The first set (training set) could be downloaded by participants for testing during development. The second set (validation set) was kept private, but could be used for testing

through the competition's website (Perez, 2016). The final set (test set) was used to determine the rankings in the actual competition. This means that the games that were finally used to rank the participants in the competition were completely unknown to the participants and never used in any way before the competition. Many of the participants in 2014 used MCTS.

In 2015, the second edition of the competition was held and associated with three conferences; ACM GECCO, IEEE CIG and IEEE CEEC (Perez-Liebana *et al.*, 2016). MCTS was still used by many of the participating agents, but it was less dominant. Some of the highest-ranking agents chose between a variety of approaches based on observations made during gameplay. These approaches include Breadth First Search, evolutionary algorithms, and A*. Another approach that performed well was Iterated Width (Lipovetzky and Geffner, 2012), which was used by the NovTea (4^{th} place at CIG) and YBCriber (1^{st} place at CEEC) agents.

In 2014 and 2015, the GVGP Competition consisted only of the *Planning track*. In this track, players have access to a *forward model* which can be used, for instance, for lookahead search. In 2016, it is planned to also run different tracks with different rules or goals; a *Two-Player Planning Track*, a *Learning track* and a *Procedural Content Generation track*. This thesis focuses on the *Planning track*.

1.3 Problem Statement and Research Questions

MCTS performed well in the GVGP Competition of 2014, and is also known to frequently perform well in related domains, such as GGP (Björnsson and Finnsson, 2009). However, in 2015, it was outperformed by several different techniques, such as Iterated Width (IW) and A*-based approaches. IW-based agents appeared to have the best performance among these approaches. In comparison to other techniques used in GVGP, IW is a relatively new algorithm (Lipovetzky and Geffner, 2012). The fact that IW outperformed MCTS in 2015 can be considered surprising, because IW is originally a planning technique, and not a game-playing technique. The **problem statement** of this thesis is:

How can Monte-Carlo Tree Search be enhanced to perform competitively in General Video Game Playing?

One way in which this problem statement can be addressed is to investigate the weaknesses of MCTS in comparison to IW for GVGP, and find ways to deal with those weaknesses. This leads to the following **research question**:

1. *How can ideas from Iterated Width be integrated in Monte-Carlo Tree Search and enhance its performance in General Video Game Playing?*

There already is a wide variety of enhancements for the MCTS algorithm, some of which are known to work well in other domains, and some of which have been proposed specifically for GVGP. To improve the performance of MCTS in GVGP, some of these enhancements are evaluated and extended, and combined with enhancements found by answering the previous research question. This leads to two more **research questions:**

2. *How can enhancements known from other domains be used to improve the performance of Monte-Carlo Tree Search in General Video Game Playing?*

3. *How can enhancements previously proposed for General Video Game Playing be extended and used to improve the performance of Monte-Carlo Tree Search?*

Additionally, it is investigated if the performance of MCTS in GVGP can be improved further with novel enhancements. This leads to the last **research question**:

4. *How can the performance of Monte-Carlo Tree Search in General Video Game Playing be improved with novel enhancements?*

1.4 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 describes General Video Game Playing (GVGP) in more detail. The competition rules, the framework used for implementing agents, and important properties of the games supported by the framework are described. Chapter 3 describes two major search techniques. It describes Monte-Carlo Tree Search (MCTS), which is the algorithm that this thesis aims to enhance, and Iterated Width (IW), which is used as inspiration for some enhancements. Chapter 4 describes the enhancements for MCTS in GVGP that are evaluated in this thesis. Chapter 5 describes the setup and results of the experiments that have been carried out to assess the enhancements for MCTS. Chapter 6 concludes the thesis by addressing the research questions and problem statement, and providing ideas for future research.

Chapter 2

General Video Game Playing

In this chapter, the concept of General Video Game Playing (GVGP) is described in detail. First, the rules of the GVGP competitions are explained. Next, the framework that runs games for the competitions is discussed. Finally, the chapter contains an analysis of the games in this framework.

2.1 Competition Rules

In the GVGP competition (Perez, 2016), a set of ten games is used to evaluate and rank the participants. Every year, this set consists of new games, and the set of games is not revealed before the competition. Every game consists of five levels (describing the initial state of the game), and every agent plays every level ten times, for a total of fifty runs per game. The agent is given 1 second of processing time before every run starts, and 40 milliseconds of processing time per *tick*. A *tick* (or *cycle* or *frame*) can be thought of as being a turn in the game, where the agent is allowed to choose an action. Every run ends in a loss after 2000 ticks, but can end in a win or a loss earlier depending on the game's specific rules. Agents are not allowed to use multi-threading.

The participants of the competition are ranked as follows. For each of the ten games, the agents are sorted according to the following three criteria, listed in order of importance here:

1. **Wins:** The total number of wins achieved by the agent in a game.
2. **Score:** The average score that the agent obtained over all the runs in a game.
3. **Time:** The total amount of time, measured in ticks, that the agent spent playing all the runs of a game.

For every game, agents are awarded 25, 18, 15, 12, 10, 8, 6, 4, 2, or 1 points for ranking first place, second place, etc. in that game. Agents ranked lower than 10^{th} do not get any points for that game. The final ranking of the entire competition is obtained by adding up the points for all of the games. This system is based on the Formula 1 score system (Fédération Internationale de l'Automobile, 2016).

2.2 The GVG-AI Framework

The framework used to run games and provide information to the agent is called the *GVG-AI* framework[1]. Levels and game rules are described in text files using a Video Game Description Language (VGDL) (Ebner *et al.*, 2013; Schaul, 2013). The information contained in these files is not provided to the agent, but only used by the framework to run games. This is different from GGP (Genesereth *et al.*, 2005), where the

[1] https://github.com/EssexUniversityMCTS/gvgai

information in such files is available to the agent. The GVG-AI framework supports the definition of a wide range of different video games. All games are played in 2D game worlds, and feature an *avatar*, which is the character that executes the agent's actions. Some games are inspired by existing video games, such as the *Aliens* game which is inspired by *Space Invaders*, and some games are entirely new. Figure 2.1 depicts how the *Aliens* game is visualized in GVG-AI.

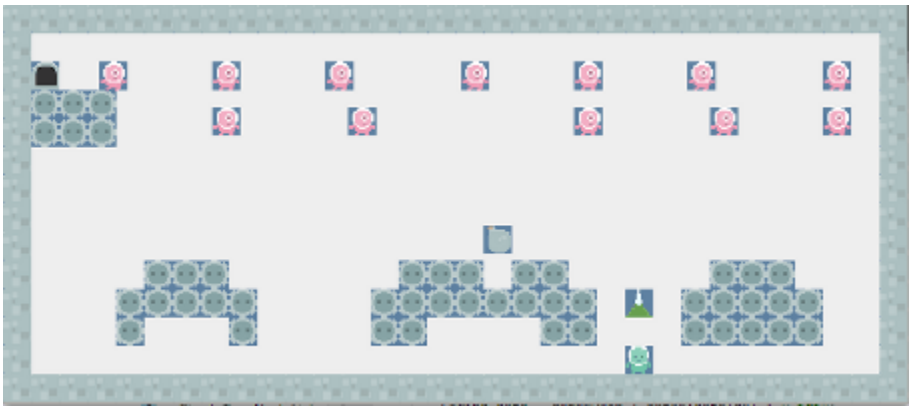


Figure 2.1: The *Aliens* game in GVG-AI. Source: http://gvgai.net.

Whenever the agent is required to decide which action to play next, it is given a `StateObservation` object that represents the current state of the game. It provides access to the following functions that the agent can use to obtain general information concerning the game state:

- `getGameScore`: Returns the score of the agent in the game state.
- `getGameTick`: Returns the duration of the run until the given game state, measured in ticks.
- `getGameWinner`: Indicates whether the game was won or lost by the agent or did not yet terminate.
- `getWorldDimension`: Returns the dimensions of the game world.
- `isGameOver`: Indicates whether or not the game terminated.

Additionally, the following functions can be used to obtain information about the avatar:

- `getAvailableActions`: Returns the list of actions that the avatar can execute in the game state.
- `getAvatarPosition`: Returns the current position of the avatar.
- `getAvatarSpeed`: Returns the current speed of the avatar.
- `getAvatarOrientation`: Returns the current orientation of the avatar.
- `getAvatarResources`: Returns a mapping from integers to integers, where the keys are IDs of resources that the avatar can have, and the values are the amounts of the resources that the avatar currently has. An example of a resource that the avatar could have in some games is ammunition.

Finally, the following functions can be used to obtain information about other objects in the game:

- `getEventsHistory`: Returns a list of all collision events between objects that have happened in previous game states. Collision events are events where something special happens when two objects of certain types collide with each other in the game world. The VGDL describes what happens when objects collide. For example, the VGDL can define that an alien is killed when it collides with a missile. This function can only tell an agent that an event occurred, and which objects were involved in the event, but not what the consequences of an event are.

- `getObservationGrid`: Returns a grid representation of the level, where every cell contains a list of the objects that are located in that cell in the game state. Each of those observations has similar functions available to obtain information as the functions for the avatar described above.
- `getXXXPositions`: There are different varieties of this function where "`XXX`" is replaced by a category of objects. The functions then return a list of objects of the corresponding category. Examples of categories are NPCs (non-playable characters), or Immovables (objects that cannot move during gameplay).

The *Forward Model* of the framework enables agents to perform lookahead searches. It consists of a `copy` function, with which any given game state can be copied, and an `advance` function, with which any given game state can be modified by simulating the effects of executing a given action in that state. A typical lookahead search would consist of copying the current game state a number of times, simulating sequences of actions on these copies of the original game state, and then evaluating the resulting game states using the functions listed above.

The framework supports six different actions that agents can choose to play; four movement actions (*Up*, *Left*, *Down* and *Right*), a *Use* action, and the *Nil* action. Playing the movement actions results in the avatar attempting to move and/or turn in the chosen direction. The exact effect of such an action depends on the game being played, and the current game state. For example, some games have physics that also affect movement, and some games can have obstacles that block movement. The *Use* action can have very different effects depending on the game being played. It can, for instance, result in the avatar shooting, or swinging a sword. Playing the *Nil* action means that the agent does not do anything. This action is also automatically played if an agent does not return an action to play in the amount of processing time that the competition rules allow. Not every action is supported in every game. For instance, the *Aliens* game does not allow the agent to move vertically (*Up* or *Down*).

2.3 Analysis of Games in GVG-AI

This section first describes properties of the games and the framework itself that are important to take into account for the development of a GVGP agent in the GVG-AI framework. Afterwards, it is discussed how the state space of these games can be modelled using game trees.

2.3.1 Properties of Games and Framework

Framework Speed

Because the framework is capable of running a large variety of different games, the games are unlikely to be implemented as efficiently as they would be in a program dedicated to only a single game. The `advance` and `copy` functions of states in this framework are therefore relatively slow. This has two important consequences. The first is that any algorithm that explores the state space of a game in GVG-AI can explore a smaller part of the state space than it would in a specialized program.

The second consequence is that enhancements of such algorithms that do not rely on generating more states have a lower relative computational overhead than similar enhancements would have in programs dedicated to single games. Consider, for example, a computationally cheap heuristic function $h_1(s)$, and a computationally expensive heuristic function $h_2(s)$, where both functions estimate the value of a state s, but $h_2(s)$ is more accurate than $h_1(s)$. Suppose that $h_2(s)$ is typically not worth using in programs dedicated to a single game, because the increased accuracy does not make up for the reduction in the number of states that can be explored due to the increased computational effort. It is possible that, in GVG-AI, the same *absolute* increase in computational cost is a smaller *relative* increase, and $h_2(s)$ can then become useful in GVG-AI.

Real-Time

Because agents are allowed to select an action to play every 40 milliseconds in the competition, the games can be considered to be *real-time* games. In comparison to abstract games, where agents typically have processing time available in the order of seconds or minutes, 40 milliseconds is a small amount of time. This means that algorithms can only explore a relatively small part of the state space of a game in GVG-AI before making a decision.

Nondeterministic

Games in GVG-AI can be nondeterministic, meaning that applying the same action in two copies of the same state does not necessarily result in the same successor state. Agents can call the `advance` function on copies of a state multiple times to handle this. A significant number of the games used in previous competitions also actually are deterministic, but this information is not directly available to agents. Some agents attempt to learn whether or not games are in fact deterministic, and switch between different algorithms based on their classification.

Fully Observable

The game states in GVG-AI are fully observable, in that the locations of all objects in the game are known to the agent. However, the rules of the game are not known. For instance, it is not known to the agent what happens when two objects collide, or under which conditions a game is won or lost. The only way to obtain this kind of information is by using the forward model to simulate the effects of actions, and observe the resulting states.

Dynamic

The games in GVG-AI can be dynamic, meaning that a game state can change even when the agent plays the *Nil* action. New objects can appear, and existing objects can disappear. Some objects, such as NPCs, can move. These movements can be deterministic or nondeterministic.

Game Score

Game states in GVG-AI have a score feature. It is important to maximize the score because it is one of the criteria used to rank agents in the competition. In some games, the score can also be used as a heuristic estimate of the value of a game state, in the same way that domain-specific evaluation functions are used in many game-specific agents.

In (Perez *et al.*, 2016), the score systems of some games are classified as *binary*, *incremental* or *discontinuous*. A binary score system means that only terminal game states have a score different from 0 (typically 1 for wins and -1 for losses). In games with this score system it is not possible to use the game score as an evaluation function for non-terminal game states. An incremental score system means that a number of events can occur in a game that all increase the score by a small value (e.g., 1 point for every enemy killed). In these games it is more reliable to use the game score as an evaluation function, especially if the winning condition of the game is also related to the same type of event (e.g., all enemies killed). A discontinuous score system means that there is a higher number of events that increase the score by a small value, as in the incremental case, but there is also a small number of other events that result in a high score increase. This can be problematic for agents that learn about the incremental score increases first and then only try to exploit those events, neglecting exploration which could result in finding the larger score increase.

Furthermore, games can have *sparse* or *delayed* scores. A game has sparse scores if there is a low number of events that lead to a score increase. All games with a binary score system are considered as having sparse scores, but games with incremental and discontinuous score systems can also have spares scores. In games with a non-binary score system, but sparse scores, it can be just as difficult to use the game score as an evaluation function as in games with a binary score system. In a game with delayed scores, it is possible that

actions lead to score increases more than one tick after playing the action. For instance, in *Aliens*, a missile launched by the *use* action needs some travelling time before it hits an object and results in a score increase. Delayed scores are typically not a problem if there is only a short delay, but they can be problematic when the delay is longer than the depth that can be feasibly reached by a search algorithm.

Single-Player

All games in the planning track of GVGP are single-player games. Some games have "enemies", in some cases with random behaviour but also sometimes with simple chasing behaviours, but these are not considered to be players. The forward model perfectly models their behaviour, which means that agents only need to take into account those moves that these opponents make according to the forward model. This is different from two-player, zero-sum games, such as chess and Go, where agents typically assume that the opponent plays perfectly, and also attempt to find the best moves that the opponent could play.

2.3.2 Game Tree Model

Tree Structure

The space searched by game-playing algorithms is typically modelled as a tree, where every node n represents a game state $s(n)$, and a directed edge $\langle n_1, n_2 \rangle$ from a node n_1 to a node n_2 represents the transition from $s(n_1)$ to $s(n_2)$ caused by playing an action $a(\langle n_1, n_2 \rangle)$. In such a tree, the root node represents the current state of the game, and leaf nodes represent terminal game states. A simple example of such a game tree is depicted in Figure 2.2. In this example, s_0 is the current game state. Playing action a_1, a_2 or a_3 in s_0 causes a transition into s_1, s_2 or s_3, respectively. If this figure depicts the complete game tree, s_1, s_2 and s_3 are all terminal game states, because they have no successors. However, if the figure only depicts a tree that was generated by a search algorithm, it is possible that these states are simply the final states that the algorithm was able to reach before running out of processing time.

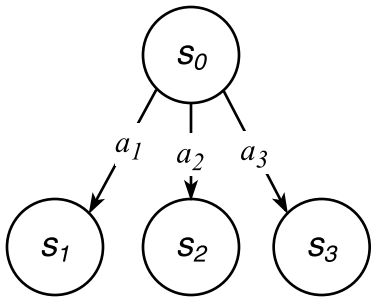


Figure 2.2: A simple example game tree.

Because games in GVG-AI can be nondeterministic, it is possible that a single action in a single state has multiple successor states. This means that a model with one edge per action in every state, and exactly one state per node, is not sufficient. Therefore, the model for nondeterministic games is typically extended with chance nodes. Chance nodes can be thought of as nodes in which the environment probabilistically "chooses" a successor whenever a state transition is nondeterministic. Such a model assumes that the probability distribution over all possible state transitions is known. This is not the case in GVG-AI, which means that it is not feasible to add chance nodes to the model.

In this thesis, the game tree in GVG-AI is modelled based on an *Open Loop* approach (Perez *et al.*, 2015). The idea of this approach is that every edge still represents an action, only the root node corresponds to a single state (the current state s_0), and every other node n represents a probability distribution over all states that can be reached by starting in s_0, and playing the sequence of actions on the edges from the root node

to n. The maximum depth of such a tree is $D = T - t$, where T is the maximum duration in ticks of a game ($T = 2000$ in the competitions), and t is the game tick of the current game state ($t = 0$ when the game starts). Most games have one of the following sets of actions available in every game state:

1. $\{Nil, Use, Left, Right\}$
2. $\{Nil, Left, Right, Up, Down\}$
3. $\{Nil, Use, Left, Right, Up, Down\}$

This means that most games consistently have the same branching factor $b \in \{4, 5, 6\}$ for every node of the tree. Many agents in past competitions ignore the fact that the *Nil* action can be played, because that action simply means that the avatar does nothing, reducing the branching factor to $b \in \{3, 4, 5\}$.

Game-Theoretic Values

Let S denote the set of all possible game states. Suppose that some state $s_T \in S$ is a terminal game state. Typically, the game-theoretic value $V(s_T)$ is then given by Equation 2.1 in single-player games.

$$V(s_T) = \begin{cases} 1 & \text{if } s_T \text{ is a winning state} \\ 0 & \text{if } s_T \text{ is a losing state} \end{cases} \tag{2.1}$$

However, in the competitions for GVGP, it is not only important to win games. As described in Section 2.1, the score is used as a tiebreaker for ranking when multiple agents have the same number of wins, and the time is used as a tiebreaker when multiple agents also have the same average score. Let $score(s_T)$ and $tick(s_T)$ denote the game score and game tick of state s_T, respectively. Then, the game-theoretic value of a terminal state s_T in GVGP is given by Equation 2.2.

$$V(s_T) = \begin{cases} M + score(s_T) - \frac{tick(s_T)}{M} & \text{if } s_T \text{ is a winning state} \\ -M + score(s_T) - \frac{tick(s_T)}{M} & \text{if } s_T \text{ is a losing state} \end{cases} \tag{2.2}$$

In this equation, M should be sufficiently large so that ties are broken correctly; $\forall s \in S(M > |score(s)|) \wedge (score(s) = 0 \vee \frac{tick(s)}{M} < |score(s)|)$. Because the different possible score values in GVGP are unknown, it is difficult to give an exact value for M, but typically a value in the order of $M = 10^7$ is sufficient.

Now let $s \in S$ denote a non-terminal game state, and let $A(s)$ denote the set of actions that can be played in s. Let $p_a(s, s')$ denote the probability that playing $a \in A(s)$ in s leads to the successor state s'. Then the expected value $\mathbb{E}(V(a, s))$ of playing an action a in s is given by Equation 2.3.

$$\mathbb{E}(V(a, s)) = \sum_{s' \in S} p_a(s, s') \times V(s') \tag{2.3}$$

Because all games in GVG-AI are single-player games, the game-theoretic value $V(s)$ of any non-terminal state s is simply the maximum expected value of the actions playable in s, as given by Equation 2.4.

$$V(s) = \max_{a \in A(s)} \mathbb{E}(V(a, s)) \tag{2.4}$$

Because nodes do not represent individual states, but a probability distribution over states, it is also interesting to look at the game-theoretic value of a node n. Let s_0 denote the root state, which is the only possible state in the root node. Let $p_n(s)$ denote the probability of observing a state $s \in S$ when applying the sequence of actions leading from the root node to n. Then, the game-theoretic value $V(n)$ of the node n, which represents the expected game-theoretic value of playing the action sequence leading to that node, is given by Equation 2.5.

$$V(n) = \sum_{s \in S} p_n(s) \times V(s) \tag{2.5}$$

Chapter 3

Search Techniques in GVGP

This chapter discusses two search techniques for decision-making in GVGP; Monte-Carlo Tree Search (MCTS) and Iterated Width (IW). Some background information that is relevant for both techniques is provided first, and the algorithms are individually described in detail afterwards.

3.1 Background

The basic idea of search techniques for decision-making in games is as follows. First, the search space of a game is modelled as a game tree, as described in Subsection 2.3.2. Then, that tree is traversed in an attempt to approximate the game-theoretic value of every child of the root node. Finally, based on those approximations, one of the children of the root node is selected, and the corresponding action is played.

Due to the limited amount of time available for decision-making in GVGP, it is not feasible to exactly compute the game-theoretic values of nodes as defined in Subsection 2.3.2. Therefore, both techniques require an evaluation function that can also provide meaningful values for non-terminal states. Such a function is then used to evaluate nodes at a depth that is feasible to reach in the available amount of time, and those evaluations are then used to approximate the values of nodes closer to the root. Unless noted otherwise, both techniques use Equation 3.1 as a basic evaluation $X(s)$ of a game state s.

$$X(s) = \begin{cases} 10^7 + score(s) & \text{if } s \text{ is a winning state} \\ -10^7 + score(s) & \text{if } s \text{ is a losing state} \\ score(s) & \text{if } s \text{ is a non-terminal state} \end{cases} \tag{3.1}$$

This evaluation function is also used by the sample MCTS agent included in the framework. For terminal states, it is similar to the game-theoretic value given by Equation 2.2. The only difference is that the $tick(s)$ term, which keeps track of the duration of a game, is not included. It has not been included because it is rarely actually relevant in the competitions; agents generally already have a difference in their average score, which means that $tick(s)$ is not used as a tie-breaker. For a non-terminal state s, the function is quite different from the game-theoretic value given by Equation 2.4; $score(s)$ is used as a heuristic evaluation of the maximum expected value of successors that cannot be investigated anymore due to computational constraints.

It is sometimes convenient if evaluations of states can be guaranteed to lie in the interval $[0, 1]$. When this is the case, the normalized evaluation of a state s is referred to as $Q(s)$ in this thesis, defined by Equation 3.2. In this equation, $\hat{X}_{min}$ and $\hat{X}_{max}$ denote, respectively, the minimum and maximum values $X(s')$ found so far for any state s' in the current game.

$$Q(s) = \frac{X(s) - \hat{X}_{min}}{\hat{X}_{max} - \hat{X}_{min}} \tag{3.2}$$

3.2 Monte-Carlo Tree Search (MCTS)

Monte-Carlo Tree Search (MCTS) (Kocsis and Szepesvári, 2006; Coulom, 2007) is a best-first search algorithm. It gradually builds up a search tree, exploring more promising parts of the search tree in more detail than less promising parts of the search tree. It uses Monte-Carlo simulations to approximate the value of game states. The algorithm is initialized with only a single node (the *root node*), which represents the current game state. Next, for as long as some computational budget allows, MCTS repeatedly executes simulations, consisting of four steps each. These four steps (Chaslot *et al.*, 2008), depicted in Figure 3.1, are:

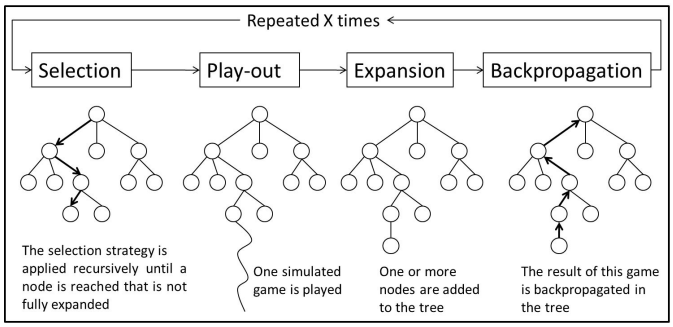


Figure 3.1: The four steps of an MCTS simulation. Adapted from (Chaslot *et al.*, 2008).

1. **Selection:** In the selection step, the algorithm traverses down the tree from the root node, according to a selection policy, until a node is reached that is not yet fully expanded. A node is considered to be fully expanded if it has one child node for every action that is available in that node. The main purpose of the remaining steps is to improve the approximation of the value of nodes traversed in this step. A good strategy for this step should provide a balance between *exploitation* of nodes that seem promising based on the results of previous simulations, and *exploration* of nodes that have not yet been visited often in previous simulations.

2. **Play-out:** In the play-out step, a (semi-)random game is played, starting from the game state in the node selected in the selection step. The purpose of this step is to rapidly obtain one of the possible outcomes of the line of play chosen in the selection step. The intuition is that the outcomes of a large number of simulations, biased by the selection strategy to contain more good lines of play than bad lines of play, can be averaged to approximate the value of the states that all the simulations have in common.

3. **Expansion:** In the expansion step, the algorithm expands the tree by adding one or more new nodes. The purpose of this step is to gradually build up an actual tree in which results can be memorized. This is the tree that is traversed by the selection step.

4. **Backpropagation:** In the backpropagation step, the outcome of the game simulated in the previous step is backpropagated to the root node. Typically this outcome is simply a value indicating whether the game was a win or a loss, but other data can be backpropagated as well if the previously described steps require it.

When the computational budget expires, the action to be played is chosen based on the outcomes of the simulations. Common strategies for this are to play the action that was selected most frequently, or to play the action leading to the node with the highest average score among the simulations. In this thesis, the average score is maximized. Pseudocode of MCTS is given in Algorithm 3.1.

There is a large number of different strategies for these four steps in existing literature (Browne *et al.*, 2012). The following subsections describe the most common ways to implement these steps, and small changes that can be applied in GVGP. Larger changes and enhancements are discussed in the next chapter.

Algorithm 3.1 Monte-Carlo Tree Search (MCTS)

Require: Root node r, root state s_0

```
1: function MCTS
2:     while computational budget not expired do
3:         (node, state) ← SELECTION(r, s_0)
4:         (node_end, outcome) ← PLAY-OUT(node, state)
5:         EXPAND
6:         BACKPROPAGATE(node_end, outcome)
7:     end while
8:     return action a leading to child c of r with highest estimated value X(c)
9: end function
```

3.2.1 Selection

The most common implementation (Kocsis and Szepesvári, 2006) of the selection step is to use a policy named *UCB1* (Auer, Cesa-Bianchi, and Fischer, 2002). An MCTS implementation with this policy is typically referred to as *Upper Confidence Bounds applied to Trees (UCT)*. Given a current node P, with successors $Succ(P)$, this policy selects the node $S_i \in Succ(P)$ that maximizes Equation 3.3.

$$UCB1(S_i) = \overline{Q}(S_i) + C \times \sqrt{\frac{ln(n_P)}{n_i}} \tag{3.3}$$

In this equation, $\overline{Q}(S_i)$ denotes the mean of all the scores that have been backpropagated through S_i, n_P denotes the number of times that P has been visited, and n_i denotes the number of times that S_i has been visited. C is a parameter, where higher values for C lead to more exploration, and lower values for C lead to more exploitation. $\overline{Q}(S_i)$ should be normalized to lie in the interval $[0, 1]$, because consistently having the same range of values for that term makes it easier to tune the C parameter.

3.2.2 Play-out

The most common implementation of the play-out step is to repeatedly play random moves (drawn uniformly from the set of available moves in every state) until a terminal game state is reached. Such an entirely uninformed selection of moves to play is computationally cheap, which typically allows for a large number of simulations, but individual simulations are unlikely to represent realistic lines of play. This means that a high number of simulations is required to obtain a reasonable approximation of the value of a state.

In the GVG-AI framework, it is not feasible to always continue simulations until a terminal game state is reached. Therefore, a depth limit is introduced, and a simulation is stopped when that depth limit has been reached (or when a terminal game state is reached earlier). The sample MCTS controller included in the framework runs simulations for at most 10 ticks after s_0, where s_0 is the state in the root node. The implementation of MCTS described in this thesis simulates at most 10 ticks after s_i, where s_i is the state in the node selected in the selection step.

3.2.3 Expansion

When a node that is not yet fully expanded is selected, the most common implementation of the expansion step is to simply add one new node to the tree. That node is typically a node chosen according to the play-out strategy, where the possible choices are nodes that should still be added to the selected node. With this strategy, every iteration of MCTS adds exactly one node to the tree.

Another option is to add one node to the tree for *every* action played in the play-out step, instead of only adding a node for the first action played in the play-out step. This is typically not done because, with a high number of simulations, this can easily lead to running out of memory. Due to the limited amount of

processing time available, and the high amount of overhead from the GVG-AI framework, the risk of running out of memory is low in GVGP. Creating and storing nodes is computationally cheap in comparison to the expensive functions of the framework. To keep as much information as possible from executed simulations, the implementation of MCTS described in this thesis simply adds one node to the tree for every action played in the play-out step.

3.2.4 Backpropagation

The backpropagation step should be implemented to backpropagate any data that the other three steps of MCTS require up the tree, starting from the end of the simulation and going back up to the root. For the most basic implementations of MCTS, this typically means keeping a sum of scores X_i and a visit count n_i in every node. Then, the evaluation $X(s)$ of the final state s of the simulation is added to the sums of all the nodes that occurred in the iteration, and the visit count of every node in the same set of nodes is incremented. These variables are required for the computation of UCT values in Equation 3.3.

3.3 Iterated Width (IW)

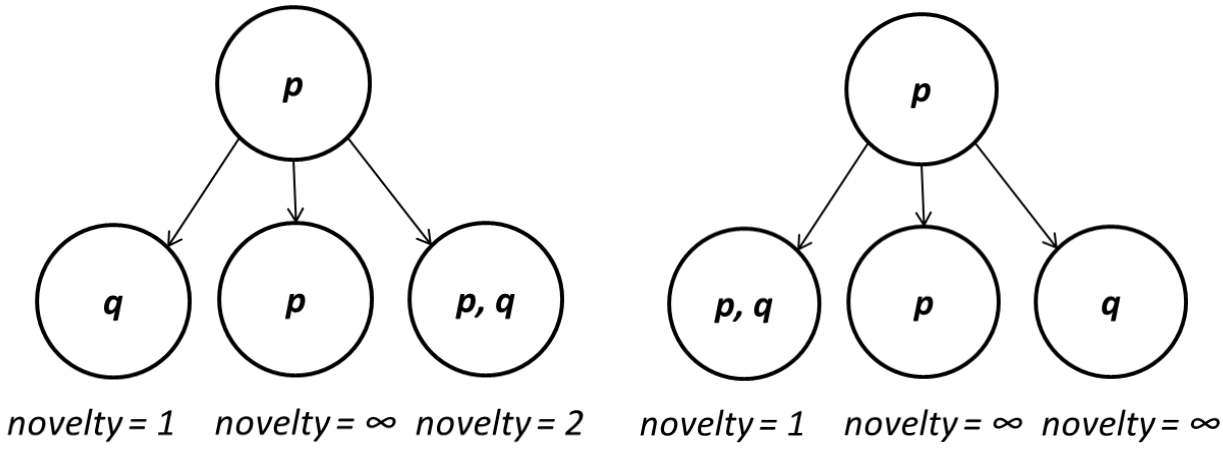
Iterated Width (IW) is an uninformed search algorithm that originates from classic planning (Lipovetzky and Geffner, 2012). The algorithm consists of a sequence of calls to IW(i) for $i = 0, 1, 2, \ldots$, until some stopping condition is reached. Examples of stopping conditions are running out of processing time, or finding a solution for a given problem. Every call to IW(i) is a Breadth-First Search process where a generated state s is pruned if s does not pass a so-called *novelty test*. A novelty test is passed during an IW(i) call if and only if $novelty(s) \leq i$, where $novelty(s)$ is the "*novelty measure*" of s.

The definition of the novelty measure $novelty(s)$ of a state s assumes that s can be defined as a set of boolean atoms that are true in that state. This is indeed the form in which states are conventionally defined in classic planning, but not in game AI. However, as described later in this section, this does not turn out to be a problem in GVGP. The definition of $novelty(s)$ is as follows:

Definition 1 *The novelty measure novelty(s) of a game state s generated by an IW(i) search process is the size of the smallest tuple of atoms that are all true in s, and not all true in any other state that has been previously generated in the same IW(i) search process.*

This means that, if in s some new atom is true that was not true in any previously seen state, $novelty(s) = 1$. If $novelty(s) \neq 1$, but a *pair* of atoms is true in s where that specific combination of two atoms was not true in any previously generated state, $novelty(s) = 2$, etc. The lowest possible novelty measure a state can have with this definition is 1, and the highest possible novelty measure is n, where n is the maximum number of atoms that can exist in any state. If s is an exact copy of a previously generated state, the novelty measure is undefined because there is no tuple of atoms that are all true in s and not all true in any previously generated state. This thesis uses the convention that $novelty(s) = \infty$ in such a case.

Two example search trees are depicted in Figure 3.2. In the first tree (Figure 3.2a), only the atom $\boldsymbol{p}$ is true in the root state. The first successor has a novelty measure of 1, because there is a tuple of size 1 (containing the atom $\boldsymbol{q}$) that is true and was not true in any previous state. The second successor has a novelty measure of ∞, because there is no tuple of atoms that are not all true in any previous state (only $\boldsymbol{p}$ is true, and that atom was already true in the root state). The third successor has a novelty measure of 2, because the smallest tuple of atoms that are all true in this state and not all true in any previously generated state has a size of 2 (a tuple containing $\boldsymbol{p}$ and $\boldsymbol{q}$). Note that $\boldsymbol{p}$ and $\boldsymbol{q}$ were both individually true in previously generated states, but not yet together in the same state. The second search tree (Figure 3.2b) contains the same states, but generated in a different order. This shows how the order in which states are generated can have a big impact on the IW algorithm. An IW(2) process, which prunes any state s with $novelty(s) > 2$, would prune only one successor in Figure 3.2a, but prune two successors in Figure 3.2b, even though exactly the same states are generated.



(a) Example search tree.

(b) The same game states generated in a different order.

Figure 3.2: Two example search trees with novelty measures. The states are assumed to be generated in a Breadth-First order. For every state, the atoms that are true in that state are inside the circle.

Intuitively, states with lower novelty measures can be seen as being "more novel", in that they are more different from previously seen states. In a call to IW(i), a state s with $novelty(s) > i$ is pruned because it is assumed to be too similar to previously seen states, and therefore not interesting to explore further. Note that only states s with $novelty(s) = \infty$ are truly duplicate states that can be safely pruned, and in general IW(i) is not complete because it can prune states too aggressively. Calls to IW(i) with lower values for i prune more aggressively than calls with higher values for i. More aggressive pruning increases the depth that can be reached in a limited amount of processing time, but also increases the risk of pruning states that would in fact be interesting to explore.

3.3.1 IW in Real-Time Games

Even though IW originates from classic planning, it has also been used for playing real-time games in the Arcade Learning Environment (ALE) and in the GVG-AI framework. Whereas in classic planning IW is used to find a goal state, in these real-time games it is used to find a promising game state (either a winning state, or a state with a high score) in a short amount of processing time, so that the agent can play the action leading to that state.

IW in the Arcade Learning Environment (ALE)

IW(1) was shown to have a state-of-the-art performance in the Atari 2600 games of ALE by Lipovetzky, Ramirez, and Geffner (2015), outperforming UCT in the majority of the games. Note that in this case only IW(1) was used, which is only a single iteration of the full IW algorithm. It is not feasible to run multiple IW(i) iterations in such a real-time environment, and an IW(2) iteration was reported not to improve much on a plain breadth-first search. Lipovetzky *et al.* (2015) also describe a best-first search algorithm named 2BFS. This algorithm uses two priority queues to store nodes for processing, where nodes in one queue are ordered by novelty measure, and nodes in the other queue are ordered by the reward accumulated so far. The algorithm then alternates between the two queues for choosing which node to process next. This algorithm only uses the novelty measure for ordering, and not for pruning. 2BFS was reported to play slightly above the level of UCT, but below the level of IW(1) on average.

IW in General Video Game Playing (GVGP)

IW(1) was also shown to outperform MCTS on average over three sets of different GVGP games each (Geffner and Geffner, 2015). As described above for the real-time games in ALE, it is infeasible to run the complete IW algorithm with the limited amount of processing time in GVGP. Geffner and Geffner (2015) also describe experiments with IW(2), and another variant, named IW($\frac{3}{2}$). Where IW(2) considers *all* pairs of atoms, IW($\frac{3}{2}$) only considers *some* pairs of atoms. More specifically, it considers only those pairs of atoms where at least one of the atoms describes information about the avatar. The intuition behind this variant is that the state of the avatar is considered to be the most important feature of the game state. IW($\frac{3}{2}$) prunes fewer nodes than IW(1), but more nodes than IW(2). IW(2) and IW($\frac{3}{2}$) were found to perform worse than IW(1) in the first two sets. However, the playing strength of IW(2) and IW($\frac{3}{2}$) scaled better than IW(1) with increases in processing time per frame in the third set. IW($\frac{3}{2}$) performed better than IW(2) in general.

Safety Prepruning

IW(i) as described above does not take into account that state transitions in GVGP can be nondeterministic; the underlying Breadth-First Search generates only a single successor state for every action in every expanded state. In the experiments described above for IW(i) in GVGP, *safety prepruning* (Geffner and Geffner, 2015) was used to prune actions that are likely to lead to an immediate game loss from the set of actions available in the current game state. This works as follows. Let s denote the current game state, and let $A(s)$ denote the set of actions available in s. Then, for every action $a \in A(s)$, M successor states s' are generated by applying a to M copies of s. The number of times where s' is a losing game state is counted and denoted by $D(a, s)$, where $0 \leq D(a, s) \leq M$. Only those actions a with the minimum value for $D(a, s)$ are considered safe and explored in the search process, and all other actions are pruned. Geffner and Geffner (2015) use $M = 10$.

3.3.2 Implementation IW in GVGP

There are two implementation details for using IW in GVGP that are important to discuss. The first is the choice of boolean atoms to represent the game state in novelty tests, and the second is the computation of the novelty measure of a game state.

State Representation

Geffner and Geffner (2015) describe that the number of boolean atoms in GVGP tends to be too large, even for IW(1), if all the features of states provided by the GVGP framework are used to construct boolean atoms. Instead, they propose to use only the following atoms to describe a game state s; $avatar(cell, orientation, atype)$, and $at(cell, stype)$. An atom $avatar(cell, orientation, atype)$ is true in s if and only if, in s, the avatar is positioned in the cell denoted by *cell*, has the orientation denoted by *orientation*, and is of the type denoted by *atype*. An atom $at(cell, stype)$ is true in s if and only if, in s, a sprite of the type denoted by *stype* is located in the cell denoted by *cell*.

Intuitively, this means that game states are characterized only by which types of objects occupy which cells (and, in the case of the *avatar*, also the orientation is taken into account). There are other possible features that could be considered as well, such as the amount of ammo that a character has, or the number of sprites of the same type that occupy the same cell (this number can be greater than 1). These features are not taken into account in order to reduce the computational cost of computing the novelty measures, which is described in more detail below. Excluding these features means that there is an increased risk of states being "incorrectly" pruned.

Computation Novelty Measure

As described above, the novelty measure $novelty(s)$ of a game state s is defined as the size of the smallest tuple of atoms that are all true in s, and not all true in any other state that has been previously generated in

a search process. Let $H(s)$ denote the set of all states that have been previously generated (the history), and let $A^i(s)$ denote the set of tuples of boolean atoms of size i that are true in s. For example, $A^1(s)$ denotes the set of 1-tuples of atoms that are true in s, and $A^2(s)$ denotes the set of pairs of atoms that are true in s.

To memorize all the tuples of atoms that have been previously seen in states $s' \in H(s)$, a number of sets $M^1, M^2, \ldots, M^n$ is used, where n is the highest possible number of atoms that can be true in any given state. Before $novelty(s)$ can be computed, all the sets M^i must already contain the atoms $a \in A^i(s')$ for all the states $s' \in H(s)$. This can be done using the `memorize` function in Algorithm 3.2, after which the `novelty` function can be used to compute $novelty(s)$.

Algorithm 3.2 Computation Novelty Measure

```
 1: function MEMORIZE(s)
 2:     n ← |A^1(s)|
 3:     for i ← 1, 2, ..., n do
 4:         for tuple t ∈ A^i(s) do
 5:             M^i ← M^i ∪ {t}
 6:         end for
 7:     end for
 8: end function

 9: function NOVELTY(s)
10:     n ← |A^1(s)|
11:     for i ← 1, 2, ..., n do
12:         for tuple t ∈ A^i(s) do
13:             if t ∉ M^i then
14:                 return i
15:             end if
16:         end for
17:     end for
18:     return ∞
19: end function
```

Let $A(s)$ denote the set of atoms that are true in s. Then, the nested loops in the two functions essentially loop through the set $(\mathcal{P}(A(s))\backslash\varnothing)$, where $\mathcal{P}(A(s))$ denotes the power set of $A(s)$. This set consists of $2^{|A(s)|}-1$ elements. Under the assumption that the sets M^i are implemented using hash tables, which support the $\cup$ and $\notin$ operators in constant time, these functions have a time complexity of $O(2^{|A(s)|})$, where $|A(s)|$ can be large in games with many sprites. These functions are also called frequently in IW(i); `novelty` is called once for every state that is generated, and `memorize` is called once for every state that is not pruned.

However, for the novelty tests of an IW(i) call, it is not necessary to be able to compute novelty measures greater than i; it is only necessary to differentiate between $novelty(s) \leq i$ and $novelty > i$. Because, considering the results found by Geffner and Geffner (2015), IW(i) calls with $i \geq 3$ are unlikely to be useful in GVGP, the pseudocode in Algorithm 3.2 can be changed to limit the value of n to a maximum of 2. This reduces the time complexity of the `memorize` and `novelty` functions to $O(|A(s)|^2)$, at the cost of losing the ability to compute the exact novelty measures of a state s with $novelty(s) > 2$. This does not matter in calls to IW(1) and IW(2). Similarly, it is also possible to create a dedicated (and even more efficient) implementation only for IW(1), which limits the value of n to a maximum of 1.

Chapter 4

Enhancements for MCTS in GVGP

Nine enhancements for MCTS in GVGP are discussed in this chapter. The first section explains three new enhancements that are partially inspired by IW. The second section describes six other enhancements, which are not necessarily inspired by IW. Some of these enhancements are known from existing research in other domains, but newly introduced to the domain of GVGP in this thesis. Other enhancements are based on previous work in GVGP, but are extended in this thesis.

4.1 Enhancements Inspired by IW

The IW(1) algorithm can loosely be described as a Breadth-First Search (BrFS) algorithm with pruning using novelty tests. Geffner and Geffner (2015) showed that the sample MCTS controller of the GVG-AI framework could be outperformed not only by the IW(1) algorithm, but also by a straightforward BrFS with safety prepruning (but no novelty tests). This implies that, at least in some games in GVGP, BrFS has advantages over MCTS. IW(1) was shown to perform better than the BrFS with only safety prepruning, indicating that pruning using novelty tests can also increase the performance.

In this section, three enhancements for MCTS are discussed that are partially inspired by IW(1) and the results mentioned above. The first two are named *Breadth-First Tree Initialization and Safety Prepruning*, and *Loss Avoidance*. They insert shallow BrFS processes in MCTS in specific situations where they are found to be beneficial. The third enhancement, named *Novelty-Based Pruning*, introduces the novelty tests of IW to MCTS.

4.1.1 Breadth-First Tree Initialization and Safety Prepruning (BFTI)

In some games in GVGP, the number of MCTS simulations that can be executed in a single 40-millisecond tick can be very low; in some extreme situations even smaller than the number of actions available in the current game state. In such situations, where every action is only evaluated based on a low number of (semi-)random MCTS simulations (or some actions are not evaluated at all), MCTS behaves nearly randomly.

This problem could be addressed by significantly lowering the depth at which play-outs are terminated, but that would in turn reduce the performance of MCTS in games where it *is* feasible to run a larger number of longer simulations. IW(1) does not suffer from this problem, because its Breadth-First Search (BrFS) will at least generate one node for every action available in the current game state if possible (in cases where even this is not possible, it is unlikely that any search algorithm can perform well). *Breadth-First Tree Initialization* (BFTI) is proposed as an enhancement for MCTS that addresses the problem described above by initializing the search tree with a 1-ply BrFS, and only starting the normal MCTS process afterwards if there is still time left. The idea of *safety prepruning* (Geffner and Geffner, 2015) is also included to deal with nondeterminism.

Let s_0 denote the current game state, with a set $A(s_0)$ of available actions. At the start of a tick, before starting the normal MCTS process, every action $a \in A(s_0)$ is used to advance up to M copies of s_0 by one tick. The reason for generating M states per action, instead of 1 (assuming that $M > 1$), is to deal with nondeterminism. Safety prepruning is done in the same way as in IW, by counting for every action how often it led to an immediate loss, and only keeping those actions that led to the minimum observed number of losses. An example of this process is depicted in Figure 4.1.

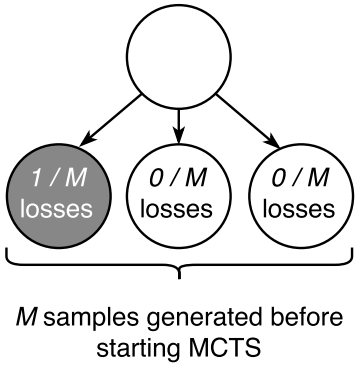


Figure 4.1: An example of Breadth-First Tree Initialization and Safety Prepruning. Out of the M states generated for the grey node, one loss was observed, whereas 0 losses were observed in the white nodes. Therefore, the grey node is pruned.

This process enables MCTS to avoid immediate losses, if possible, in situations where it might otherwise randomly select an action leading to an immediate loss due to being unable to sufficiently evaluate all available actions. Additionally, for every action $a \in A(s_0)$, the M successor states are evaluated and the average evaluation out of M evaluations is stored in the node that a leads to with a weight equal to a single normal MCTS simulation. This means that, if there is only time left for a few MCTS simulations after BFTI, MCTS can prioritize those actions that immediately lead to high average evaluations.

M is a parameter where high values for M lead to an increased certainty in nondeterministic games that immediate game losses are avoided, but also a reduced amount of time for subsequent MCTS simulations which are still necessary for long-term planning. For every action a, the M generated states corresponding to that action are cached in the node that a leads to, so that each state can be used for one MCTS simulation going through the same node after BFTI. If there is sufficient time left for MCTS simulations after BFTI, this can reduce the negative effect that the computational overhead of BFTI has on the number of MCTS simulations.

4.1.2 Loss Avoidance (LA)

The game trees of some games in GVGP have a high number of losing game states, and only "narrow paths" leading to winning game states. An example of such a game is the *Frogs* game, where the goal is to cross a road with harmful trucks and cross a river by jumping on logs floating in that river. A simplified example of a game tree in such a game is depicted in Figure 4.2. There are many losing game states (dark nodes), because there are many actions that lead to collision with trucks or the river. There is a path of white nodes which can be viewed as a narrow path leading to victory by careful movement. The rightmost action in the root node can be viewed as an action that does not significantly change the game state (for instance, an action where the player stays on the original side of the map and does not move towards the road). However, if this action is repeatedly applied it will lead to a loss due to reaching the maximum allowed duration of a game.

If the MCTS algorithm is used in this game tree with a (semi-)random play-out strategy, it is possible to get a pessimistic initial evaluation of all actions because it is likely that most of the simulations end in losses. In a game tree like this, these overly pessimistic evaluations are only corrected once the majority of the nodes

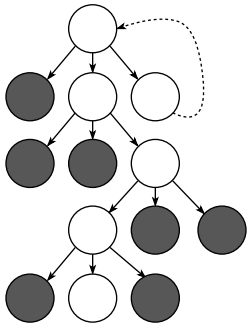


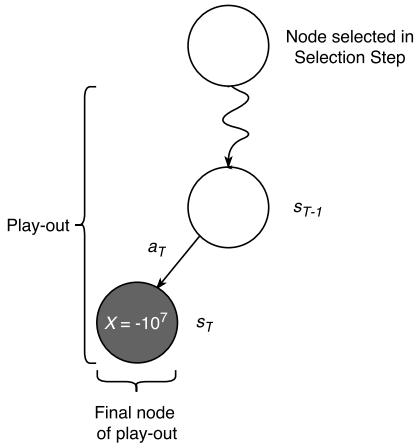
Figure 4.2: Example search tree with many losing game states. Dark nodes represent losing game states, and white nodes represent winning or neutral game states. The dashed arrow pointing towards the root node indicates that a similar state is reached.

have expanded through the expansion step, because then the selection step of MCTS can bias simulations to go through the correct nodes more frequently than the losing nodes. When this does not happen due to a lack of time in the real-time GVGP games, MCTS is most likely to play the rightmost action in the root node, because this action has a lower chance than the others of ending in a loss through (semi-)random play.

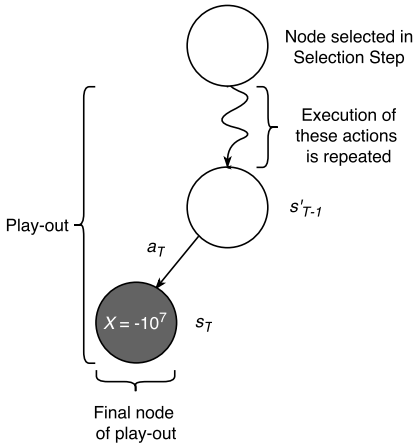
This problem is closely related to the problem of game trees with *traps* (Ramanujan, Sabharwal, and Selman, 2010) or *optimistic moves* (Finnsson and Björnsson, 2011) in (two-player) adversarial games. These are situations where MCTS gains an overly optimistic evaluation of actions that appear to lead to wins through (semi-)random plays, but actually result in states where the opponent can force a loss (a win for the opponent) with optimal play. In such situations, the use of shallow minimax searches in various steps of MCTS has been found to be beneficial (Baier and Winands, 2015), because minimax is more likely to find narrow, optimal lines of play than the (semi-)random play-outs of MCTS.

It is difficult to use minimax in a similar way to *prove* wins or losses in GVGP because the games can be nondeterministic. However, to address the problem of overly pessimistic state evaluations, it is sufficient to find alternative actions that do not lead to immediate losses when a loss is encountered through (semi-) random play in a simulation. This can be achieved by inserting a shallow BrFS, instead of a shallow minimax search. This enhancement is referred to as *Loss Avoidance* (LA).

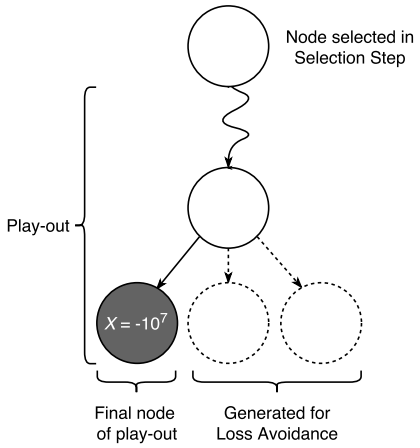
An example of how this works is depicted in Figure 4.3. Let s_T denote a losing game state at the end of an MCTS play-out, with a parent state s_{T-1}. Let a_T denote the action that led from s_{T-1} to s_T. In a regular MCTS simulation, the evaluation $X(s_T)$ of s_T would be backpropagated, which no longer happens with LA if s_T is a losing state. Instead, the idea is to generate all the siblings of the node in which s_T was observed, and generate one state for each of those nodes. Those states are also evaluated, and the best evaluation among all the siblings is backpropagated as if it was the result of the original play-out. This is only done if the losing state s_T was generated through the (semi-)random play of the play-out step. If the losing state is already encountered in the selection step of MCTS, it is backpropagated as normal and LA is not applied.



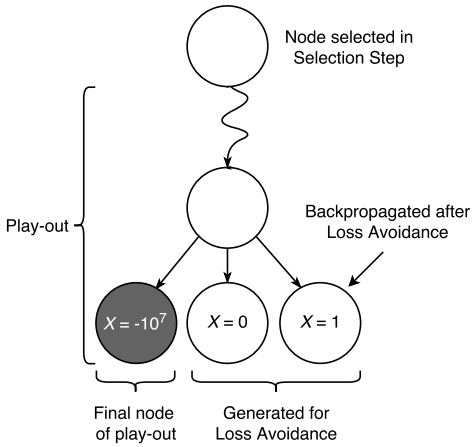
(a) A regular MCTS play-out ends in a loss.



(b) All actions played in the simulation, except for the last one, are repeated.



(c) The siblings of the final node of the original play-out are generated.



(d) The generated nodes are evaluated and the highest evaluation is backpropagated.

Figure 4.3: Example MCTS simulation with Loss Avoidance. The X values are evaluations of game states in those nodes. The dark node is a losing node.

Because the GVG-AI framework does not support an *undo* operator to revert back to an earlier game state, this cannot be implemented by reverting back to s_{T-1} from s_T, and applying all the actions available in s_{T-1} (other than a_T) to copies of s_{T-1}. This means that the entire sequence of actions that led from the root state s_0 to the state s_{T-1} must be applied again to a new copy of s_0. Let s'_{T-1} denote the resulting state (which may be different from s_{T-1} in nondeterministic games). As many copies as are necessary can be created of s'_{T-1}, and then advanced using all the available actions except for a_T.

This implementation has two important consequences. The first consequence is that it is possible in nondeterministic games that a terminal state is encountered before the entire action sequence has been executed (i.e., closer to the root node than the original final node). To avoid spending too much time evaluating a single MCTS simulation by recursively starting LA processes, the evaluation of such a state is immediately backpropagated. The second consequence is that it has a significant computational cost. The entire LA process has roughly the same computational cost as the original simulation did. This means that in a game like *Frogs*, where almost every simulation ends in a losing game state, the number of MCTS simulations that can be done per tick is roughly halved.

4.1.3 Novelty-Based Pruning (NBP)

Background

The intuition behind pruning nodes based on the results of *novelty tests* in Iterated Width (IW) (Lipovetzky and Geffner, 2012), as described in Section 3.3, is to prune states that are similar to previously seen states. Because IW is based on Breadth-First Search (BrFS), a state s that is generated earlier than another state s' can be reached by playing a sequence of actions from the root state that is at most as long as the path from the root state to s' (and possibly shorter). If s' has a high novelty measure ($novelty(s') > i$), it is pruned by IW(i) because it is considered to be a redundant state; similar states are already known to be reachable by playing different sequences of actions that are at most as long, or possibly shorter. Such prunings are not guaranteed to be "correct", or safe, and can cause states to be incorrectly pruned. However, because IW(1) was found to outperform a BrFS in GVGP (Geffner and Geffner, 2015), they can still be considered to be beneficial on average.

In this subsection, it is proposed to reduce the search space by pruning some nodes based on novelty tests in MCTS. This is referred to as *Novelty-Based Pruning* (NBP). This idea is different from most existing techniques for pruning in MCTS (Browne *et al.*, 2012), because those typically aim to prune nodes that are expected to have a low value. The idea of NBP is to prune states that are similar to other states, which are therefore also expected to have a similar value (which can be a high value). This can mean that, for example, the second-best node is pruned in some situations. The goal of such a pruning is to make it more likely that a large number of MCTS simulations can be executed to evaluate the best node, instead of splitting MCTS simulations between the two best nodes.

In some sense, this idea can be considered to be more closely related to dealing with transpositions in MCTS (Childs, Brodeur, and Kocsis, 2008), than it is related to other pruning techniques in MCTS. Typically, when *transpositions* (identical game states that can be reached by different paths in the game tree) are recognized, statistics gathered for those states are shared in some way. The idea of NBP can be viewed as using a relaxed definition of "equality" when identifying transpositions, and pruning all but one of all the "equal" states, instead of sharing results among them.

It is not trivial to directly use novelty tests in MCTS in the same way as they are used in IW. For instance, the original definition of the novelty measure $novelty(s)$ of a state s (Definition 1 in Section 3.3) assumes that states are generated in a Breadth-First order when referring to "any other state that has been previously generated". MCTS generates states in a very different order. Consider, for example, that the last state generated in the first MCTS simulation is likely much deeper in the tree than the first state generated in the second simulation. This means that the definition of the novelty measure needs to be adjusted for use in MCTS. This, in turn, has implications for the implementation (and efficiency) of the computation of novelty measures. These issues are discussed next, after which it is described exactly how the novelty tests are used for pruning in MCTS. Note that, for simplicity, the following discussion initially does not differentiate between states and nodes. This means that the initial discussion is only applicable to game trees that do not use an open-loop model. This issue is addressed later in the following discussion.

Definition Novelty Measure in MCTS

The original definition of the novelty measure $novelty(s)$ of a state s refers to "any other state that has been previously generated". This definition is only correct when used in a Breadth-First Search (BrFS) process, such as IW(i). For NBP, the definition is adjusted as follows, referring to a specific set named the *neighbourhood* $N(s)$ of s:

Definition 2 *The novelty measure $novelty(s)$ of a game state s is the size of the smallest tuple of atoms that are all true in s, and not all true in any state in the neighbourhood $N(s)$ of s.*

For defining $N(s)$, the following terminology is used:

- $p(s)$: The parent state (or predecessor) of a state s.
- $Sib(s)$: A set containing the siblings of a state s, that is, all successors of $p(s)$ except for s itself.

- $Sib_{Left}(s)$: A set containing all the siblings of s that are "to the left" of s. This assumes that $p(s)$ stores its successors in an ordered list, and states are considered to be to the left of s if they have a lower index than s in this list. Note that this means that the order in which successors are stored matters, in the same way that the order in which successors are generated matters in IW, as seen in Figure 3.2.

The following properties are considered to be desirable for any possible definition of $N(s)$:

- $N(s)$ should be a subset of the set of states that would be included in IW. If more states are included (for instance, states deeper in the tree than s), states that are far away from the root could be incorrectly prioritized over states closer to the root.
- $N(s)$ should contain as many states as possible, without violating the property described above. As the size of $N(s)$ increases, the ability for NBP to prune nodes also increases.
- Using the definition of $N(s)$, it should be possible to compute $novelty(s)$ efficiently.

$N(s)$ as used for NBP in this thesis is defined as:

$$N(s) = \begin{cases} \varnothing & \text{if } s \text{ is the root state} \\ Sib_{Left}(s) \cup \{p(s)\} \cup Sib(p(s)) \cup N(p(s)) & \text{otherwise} \end{cases}$$

An example using this definition is depicted in Figure 4.4, where all grey states are in $N(s)$. This is not the

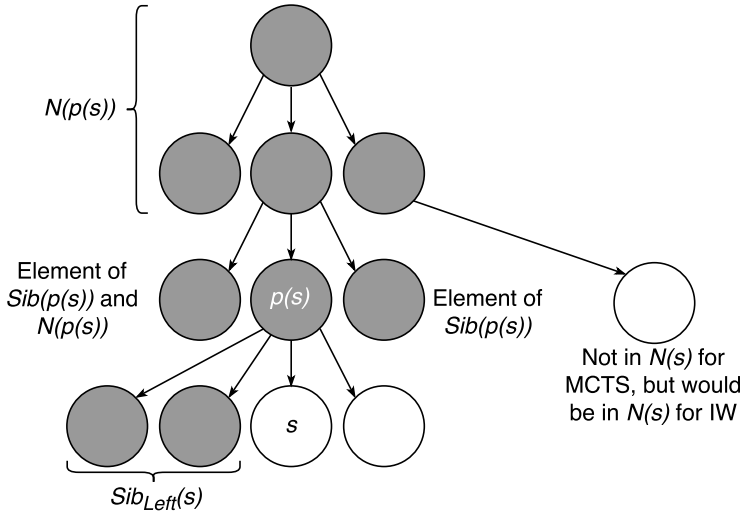


Figure 4.4: Example tree where the states in $N(s)$ are grey.

only way in which $N(s)$ can be defined. The main reason for choosing this definition is that it was considered to provide a good trade-off between the desirable properties listed above. The implementation of computing $novelty(s)$ using this definition of $N(s)$ is described next, followed by an explanation of why this is considered to provide a good trade-off between the desirable properties.

Novelty Tests in MCTS

In Section 3.3, it is explained how $novelty(s)$ can be computed efficiently in IW(i). Basically, whenever a state s is generated, all tuples of atoms that are all true in s are generated. Then, $novelty(s)$ is computed by checking which of these tuples, if any, are not yet in previously memorized sets of tuples. Finally, the

generated tuples are added to the memorized sets of tuples. For the entire search process, there is only one large collection of memorized tuples, because *all* previously generated states are considered in IW.

Because the neighbourhood $N(s)$ does not contain all previously generated states, it is also not possible to memorize all tuples in one large collection in MCTS. Some tuples of atoms should only be memorized in certain parts of the tree, and not in other parts of the tree. This is the main reason why it is not straightforward to efficiently implement novelty tests in MCTS.

In IW(i), $novelty(s)$ is computed to decide whether or not to prune s immediately when s is generated. This is done because, once s has been generated, it must be added to a First-In-First-Out queue containing states for which successors need to be generated if it is not pruned, or discarded if it is pruned. In MCTS, it is not necessary to immediately decide whether or not to prune s once it has been generated. It is possible to have a few simulations go through s before deciding whether or not to prune it. In fact, once s has been generated, the computational cost of generating that state has already been paid, so it is likely better not to prune it immediately and allow at least the current simulation to continue from s. Therefore, unlike IW, $novelty(s)$ is *not* computed as soon as s is generated. Let $Succ(s)$ denote the ordered list of successors of a state s (technically, s should be a node, but this is addressed later). The first time that s is visited in an MCTS simulation when it already is fully expanded (which is also the first time that a successor of s is chosen according to the selection policy of MCTS, instead of the play-out policy), novelty tests are performed immediately for all successors $s' \in Succ(s)$ of s.

The pseudocode for how this is done is given by Algorithm 4.1. This pseudocode is not as efficient as the implementation of novelty tests in IW, in that it is not sufficient to compare the tuples generated for a successor s' to only one large set of previously seen tuples. Instead, the novelty test is split up in multiple parts. The novelty test done in lines 7-9 corresponds to a novelty test with $N(s') = Sib_{Left}(s') \cup \{p(s')\}$, which is only a part of the chosen definition for $N(s')$. If s' already is not novel according to the test with this subset of $N(s')$, it can never be novel with the full definition of $N(s')$. However, if it is novel according to the first test, the other tests are required as well. $Sib(p(s'))$ is added to $N(s')$ in line 14 in the first iteration of the loop, and subsequent iterations of the loop also add $N(p(s'))$ to $N(s')$.

This implementation is still considered to be reasonably efficient because it only "moves" up the tree to the root, and never back "down" again (which would be necessary to, for example, take the white node into account in the right-hand side of Figure 4.4). Because the successors of a state s are only tested once s is fully expanded (and visited again in a new MCTS simulation), only states relatively close to the root tend to be tested. This means that the path up to the root, and therefore the number of separate tests that the full novelty test is split up in, also tends to be small.

The boolean atoms used for novelty tests in MCTS are predicates of the form $avatar(cell, orientation, atype)$ for the avatar, and $at(cell, stype)$ for other objects. These are the same atoms that Geffner and Geffner (2015) used for IW in GVGP, as described in Section 3.3.

Novelty Tests in Open-Loop MCTS

In the discussion above, it is mentioned that novelty tests are done for the successors $Succ(s)$ of a state s the first time that s is reached in an MCTS simulation where it already is fully expanded. It is now discussed how this can be applied in an open-loop model, where nodes in the tree can represent more than one state. The first time that a *node* n is reached in an MCTS simulation when it is fully expanded, with a state s generated in that particular simulation for n, one state s' is generated for every successor node n'. Using those particular successor states s' (all generated from copies of the same state s), the novelty tests are performed as described above. The states s' are cached in the nodes n', so that they can be re-used in subsequent MCTS simulations going through those nodes. When a novelty test for a state s' is failed, the corresponding *node* n' is marked as "not novel".

With this implementation the first part of every novelty test (performed in line 7 of Algorithm 4.1) uses a consistent set of states, in that all successors s' have the same parent state s (and not only the same action sequence leading from the root node to s). The parts of the novelty tests performed in the loop may, in nondeterministic games, compare states that are not in fact really comparable due to nondeterministic state transitions, but this is difficult to avoid.

Algorithm 4.1 Novelty Tests in MCTS

Require: A fully expanded state s with ordered list of successors $Succ(s)$, maximum novelty threshold i

```
1:  function NoveltyTestSuccessors(s)
2:      M ← ∅
3:      M_s ← GenerateTuples(s)          ▷ stores all tuples of atoms that are true (up to a size of i) in s
4:      MemorizeTuples(M_s, M)                                  ▷ stores all tuples of M_s in M

5:      for every successor s' ∈ Succ(s) do
6:          M_s' ← GenerateTuples(s')
7:          n ← ComputeNovelty(M_s', M)          ▷ size of smallest tuple that is in M_s' and not in M
8:          if n > i then
9:              mark s' not novel
10:         else
11:             p ← parent(s')
12:             while p ≠ null do
13:                 M_p ← GetCachedMemory(p)                    ▷ returns cached set of tuples
14:                 n ← ComputeNovelty(M_s', M_p)
15:                 if n > i then
16:                     mark s' not novel
17:                     break
18:                 end if
19:                 p ← parent(p)
20:             end while
21:         end if
22:         MemorizeTuples(M_s', M)
23:     end for

24:     for every successor s' ∈ Succ(s) do
25:         s'.CacheMemory(M)          ▷ caches set of tuples found in s, s' and all siblings of s'
26:     end for
27: end function
```

Pruning Nodes Marked as Not Novel

When a node n is marked as not novel, the idea of NBP is to prune n. "Pruning" a node n in MCTS is assumed to mean that n can no longer be selected by the selection policy. There are two different options that are typically considered for pruning nodes in MCTS; *soft pruning* and *hard pruning* (Browne *et al.*, 2012). Soft pruning n means that n is temporarily pruned, but may be "unpruned" again later (for instance, when the parent or siblings of n reach a certain visit count). Hard pruning n means that n is permanently pruned.

NBP uses soft pruning to prune nodes that have been marked as not novel. Nodes are not unpruned over time (as done in, for example, *Progressive Unpruning* (Chaslot *et al.*, 2008)), because this is difficult to do in a meaningful way in games with a low average number of simulations per tick. Often, such a strategy would be similar to either not pruning at all (when unpruning too quickly), or hard pruning (when unpruning too slowly). When there is only a small number of simulations, it is difficult to find a good compromise between these two extremes. Instead, a pruned node n is only unpruned when the parent node $p(n)$ appears to be a dangerous situation. The reasoning behind this is that, if $p(n)$ appears to be a dangerous situation where the likelihood of losing the game is estimated to be high, all alternatives should be considered in an attempt to find a way out of the dangerous situation. More formally, this is implemented by unpruning any pruned node n if the normalized, average score $\overline{Q}(p(n))$ of the parent $p(n)$ has a value $\overline{Q}(p(n)) < 0.5$. This unpruning is not permanent either; n can be pruned or unpruned repeatedly as $\overline{Q}(p(n))$ rises above or drops below 0.5.

When choosing the action to play at the end of every game tick, successors of the root node that are marked as not novel are treated in a similar way as described above. Nodes marked as not novel are initially ignored, and the node n_{max} that maximizes the average score among the remaining nodes is selected. However, if such a node n_{max} has a normalized average score $\overline{Q}(n_{max}) < 0.5$, and there also is a pruned node n with a higher score $\overline{Q}(n) > \overline{Q}(n_{max})$ and a visit count greater than 1, such a node is considered as well. Only pruned nodes with a visit count greater than 1 can potentially be considered, because the estimated evaluation of such a node is otherwise considered to be too unreliable.

Special Cases in Novelty Tests

There is a number of special cases concerning the results of novelty tests in MCTS that have not been covered yet by the description above. These special cases are described next.

- Whenever all successors $n' \in Succ(n)$ of a node n are marked as not novel, n itself is also marked as not novel. It is not necessary to state this explicitly for IW, because if all successors $n' \in Succ(n)$ are pruned in the Breadth-First Search of IW, the entire subtree below n is automatically pruned. However, in MCTS this should be done explicitly, because otherwise the selection policy can end up in a node in which there are only successors that are not novel.

- Let s_{t+1} denote the state generated for the novelty test of a node n_{t+1}, where s_t and n_t denote the parent state and node, respectively. Let $score(s)$ denote the game score of a state s. If $score(s_{t+1}) > score(s_t)$, node n_{t+1} is considered to be novel, regardless of the results of any novelty tests. This is done to reduce the risk of pruning actions that are directly responsible for an increase in game score. It should be noted that this risk is not entirely eliminated in games with delayed rewards. Consider, for example, the game of *Aliens*, where there can be a delay between firing a missile (an action that can be responsible for a score increase), and the missile colliding with an alien (which is the point in time where the score increase can be observed).

- Let a_{t+1} denote an action leading from a node n_t to a node n_{t+1}, where n_{t+1} is a node that would be marked as not novel according to the description above. If a_{t+1} is a movement action ($a_{t+1} \in \{Up, Left, Right, Down\}$), there are two exceptions where n_{t+1} is actually not marked as not novel. The first exception is that, if either only horizontal or only vertical movement is available in a game, n_{t+1} is not pruned. The reasoning behind this is that games where the avatar is restricted to moving along one axis, optimal play often involves moving back and forth. Such movement patterns are likely to get pruned by NBP. The second exception is that, if the avatar has a low movement speed (specifically, a speed $<= 0.5$), n_{t+1} is also not pruned. The reasoning behind this is that, with a low movement speed, the avatar often spends multiple game ticks in the same cell. This means that in multiple states in a row, the same $avatar(cell, orientation, atype)$ predicate is true, even though the avatar is in fact moving (slowly).

4.2 Other Enhancements

In this section, six other enhancements for MCTS in GVGP are discussed that are not related to IW. They do not necessarily address any weaknesses that MCTS has in comparison to IW, but are intended to improve the performance of MCTS in other ways. These enhancements are not newly proposed in this thesis, but most of them are either extended or newly introduced to the domain of GVGP in this thesis.

4.2.1 Progressive History (PH)

Progressive History (Nijssen and Winands, 2011) (PH) is an enhancement for the selection step of MCTS. The intuition of PH is to add an extra term to the UCB1 formula (Equation 3.3) that introduces a bias towards playing actions that appeared to perform well in earlier simulations, and progressively decrease the

weight of this bias as the visit count of a node increases. The selection policy is changed to select the successor $S_i \in Succ(P)$ of the current node P that maximizes Equation 4.1, instead of Equation 3.3.

$$UCB1_{ProgHist}(S_i) = \overline{Q}(S_i) + C \times \sqrt{\frac{ln(n_P)}{n_i}} + \frac{\overline{Q}(a_i) \times W}{n_i - (n_i \times \overline{Q}(S_i)) + 1} \tag{4.1}$$

In this equation, $\overline{Q}(a_i)$ denotes the mean of all the scores that have been backpropagated through edges corresponding to playing a_i, where a_i denotes the action that leads from P to S_i. W is a parameter, where a higher value for W leads to an increased weight for the Progressive History heuristic. The denominator of the new term reduces the effect of the bias for nodes that did not perform well in past MCTS simulations. This relies on $\overline{Q}(S_i)$ being normalized to lie in the interval $[0, 1]$. The $\overline{Q}(a_i)$ value is also normalized to $[0, 1]$. If there is no $\overline{Q}(a_i)$ value available for an action a_i, it is set to 1 to reward exploration of actions that have not been tried yet.

PH was previously evaluated in GVGP (Schuster, 2015) and found not to perform well. This is because the average action scores $\overline{Q}(a)$ were only collected for $a \in \{Left, Right, Up, Down, Use\}$. Such scores are unlikely to be accurate, because the true value of an action greatly depends on the state in which it is played. Therefore, the implementation of this thesis takes the cell in which the avatar is located before executing an action into account when updating or retrieving the value of $\overline{Q}(a)$. Other features of the game state are not taken into account, because they are likely less important. Including them would increase the computational cost of updating or retrieving $\overline{Q}(a)$ values, and reduce the number of observations on which every $\overline{Q}(a)$ value is based.

4.2.2 N-Gram Selection Technique (NST)

N-Gram Selection Technique (Tak, Winands, and Björnsson, 2012) (NST) is an enhancement for the play-out step of MCTS. Where PH adds a bias based on the observed quality of actions to the selection policy, NST adds a similar bias based on the observed quality of *sequences* (N-grams) of actions to the *play-out* step. This can make simulations more likely to resemble realistic lines of play than simulations using a purely random play-out policy, at the cost of increased computational cost.

NST works as follows. Let $\langle a_0, a_1, \ldots, a_n \rangle$ denote the sequence of actions that led from the root node to the final node of a simulation, where a_0 is the first action played and a_n is the last action played. A subsequence of length N of that sequence is referred to as an N-gram. For example, $\langle a_0, a_1, a_2 \rangle$ and $\langle a_{n-2}, a_{n-1}, a_n \rangle$ are both 3-grams. For every possible N-gram of actions, the average evaluation of simulations in which that N-gram occurred is stored in a table. If a specific N-gram occurs twice in the same simulation, that evaluation is counted twice when computing the average for that N-gram.

Then, in the play-out step, an ϵ-greedy strategy (Sutton and Barto, 1998) is used to select actions based on these scores as follows. With a probability of ϵ (with $0 \leq \epsilon \leq 1$), a random action is chosen. With a probability of $1 - \epsilon$, an action is chosen based on the average N-gram scores. Let a_t denote one of the actions that should be considered. Let a_{t-1} denote the action played in the previous step of the simulation, a_{t-2} the action played before that, etc. Then, $\langle a_t \rangle$ is the new candidate 1-gram, $\langle a_{t-1}, a_t \rangle$ is the new candidate 2-gram, etc. The action a_t is played if and only if it is the action that maximizes the average of the averages stored in the table for the candidate N-grams. To motivate exploration of actions that have not been tried yet, a candidate action for which no data is available yet is given the maximum score observed so far in the entire game.

N-gram scores gathered in previous ticks are likely still relevant in later ticks. Therefore, it is valuable to keep this data in memory in between game ticks. However, the scores found in previous ticks are partially based on parts of the search tree that can no longer be reached, which reduces their relevance (Tak, Winands, and Björnsson, 2014). Therefore, the scores and visit counts stored in the table for N-grams should be decayed over time. This thesis uses the *Move Decay* (Tak *et al.*, 2014) strategy, which multiplies all the scores and visit counts in the table by a factor $\gamma \in [0, 1]$ every time when the agent plays a move.

N-grams with $N > 1$ are only considered when that N-gram has been encountered at least k times (in this thesis, $k = 7$). In this thesis, only N-grams up to a size of $N = 3$ are considered. In the case of nodes

close to the root, only smaller N-grams are considered. For example, if an action needs to be selected in the root node, there is no previously played action a_{t-1}; in that case only 1-grams are considered.

Just like PH, NST has previously been evaluated in GVGP (Schuster, 2015), not taking into account the position of the avatar when storing or retrieving data for actions. In the case of NST, it was already found to provide an increase in performance, but the implementation in this thesis has still been adapted to take the location of the avatar into account in the same way as in PH. When PH and NST are combined, the data for the 1-grams of NST is the same data that is also used by PH. This means that combining the two results in a slightly smaller computational overhead than the sum of the computational overhead of the two individual enhancements.

4.2.3 Tree Reuse (TR)

After playing an action, some of the results of the past MCTS search can still be useful in the next game tick; the previous search tree can be reused. This was found to be useful in the real-time game of *Ms Pac-Man* (Pepels, Winands, and Lanctot, 2014). It has also previously been described in the context of GVGP by Perez *et al.* (2015), but they did not evaluate the impact of this idea on the playing strength on its own. This idea, named *Tree Reuse* (TR), is depicted in Figure 4.5. It works as follows. Let $t-1$ denote the previous tick, at the end of which an action a_{t-1} was played which resulted in a transition to the current state s_t with a tick counter t. In the search tree built up by MCTS during tick $t-1$, there must be an edge with the action a_{t-1} pointing from the root node to some successor node n. To start the new search process, the tree is initialized with n as the new root. The entire subtree below n is kept, and the siblings and parent of n are discarded.

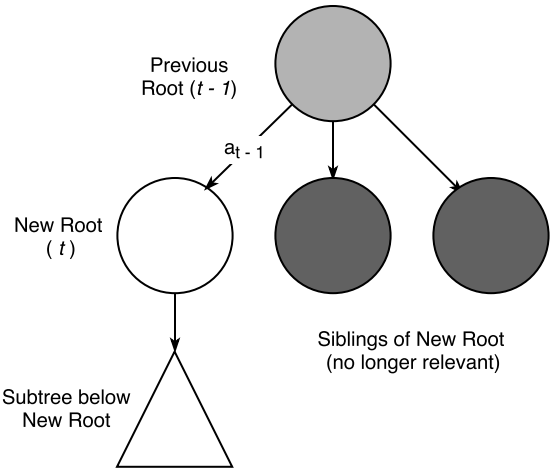


Figure 4.5: Tree Reuse in MCTS.

In nondeterministic games, the results gathered during tick $t-1$ in the new root node n are not entirely representative of the current game state. In the previous search process, where n was a successor of the root node instead of being the root node itself, it represented all possible states that could be reached by playing a_{t-1} in the previous game state. In the current tick, where n has become the root node because a_{t-1} was played in the previous tick, it is known which specific state has been reached out of the states that could have been reached, and therefore it only needs to represent a single state; the current game state. This means that the scores that were previously backpropagated into n should be given a lower weight than the scores that will be backpropagated during the current tick. The same also holds for all other nodes in the new search tree; because n now deterministically represents a single game state, all nodes in the tree below n are also likely to have a lower number of possible game states that they represent. Therefore, when a tree from a

previous tick is reused in the next tick, the total scores and the numbers of visits in every node are decayed by multiplying them with a factor $\gamma \in [0, 1]$ (Pepels *et al.*, 2014). An alternative implementation could be to memorize for every node in which tick it was last generated or decayed, and only decay a node and its successors once it is actually visited in a new tick.

When the TR and BFTI enhancements are combined, the BFTI process is still executed even if all the possible children of the new root n already exist. This is done because the safety prepruning of BFTI can still be valuable. A variant where this is not done has also been implemented and evaluated, but unless it is explicitly stated otherwise, this thesis assumes that the BFTI process runs at the start of every tick even when combined with TR.

When TR is combined with NBP, any successors of the new root node that were previously marked as not novel have this mark removed. Then, assuming that the new root is already fully expanded, the novelty tests for the direct successors of the new root are repeated in the first simulation of the new game tick. This is done because, in nondeterministic games, the results of old novelty tests may no longer be accurate. Results of novelty tests in nodes deeper in the tree than the first ply are not reset, to avoid the computational cost of repeating too many novelty tests.

4.2.4 Knowledge-Based Evaluations (KBE)

Background

In many games in GVGP, there can be situations where none of the simulations of MCTS end in game states that, according to the standard evaluation function of Equation 3.1, have a different evaluation than the current game state. This is because, due to the depth limit of the play-out step, simulations are not guaranteed to end in terminal game states, and some games also have very few or no intermediate score changes. In such situations, MCTS explores the search space randomly (because early simulations provide no extra information to guide late simulations), and also behaves randomly (because, in the end, all available actions have the same estimated value).

In the competition of 2014 (Perez *et al.*, 2016), some agents included heuristic evaluation functions taking into account estimates of the distances to various objects. A related idea was described by Perez, Samothrakis, and Lucas (2014). They proposed to keep track of the average change in score that occurs simultaneously with collision events between the avatar and other objects. This average change in score is then used in an evaluation function to reward the avatar for moving towards objects that appear to increase the score upon collision, and punish the avatar for moving towards objects that appear to decrease the score upon collision. This idea was extended in a number of ways by Van Eeden (2015). Chu *et al.* (2015) proposed to use MCTS in a similar way to learn which objects are interesting to move towards, but then use a pathfinding algorithm to move towards them (instead of a heuristic evaluation function to guide MCTS towards them). In this section, a heuristic evaluation function is proposed that is based on the work described by Perez *et al.* (2014) and Van Eeden (2015), but a number of implementation details is different. This enhancement is referred to as *Knowledge-Based Evaluations* (KBE).

Evaluation Function

The evaluation function created for this thesis is given by Equation 4.2.

$$Eval_{KB}(s_T) = \sum_i w_i \times (d_0(i) - d_T(i)) \tag{4.2}$$

In this equation, i denotes the type of an object in the current game. Object types i of objects that are created by the avatar, the type(s) of the avatar itself, and the "wall" type are not included in the sum, but all other observed types i are included. w_i denotes a weight that is adjusted during gameplay based on observations made during MCTS simulations. Intuitively, a weight w_i is increased if it appears to be good to move towards objects of the type i, and it is decreased if it appears to be bad to move towards objects of type i. A more detailed description of how the weights are determined can be found below. s_T denotes

the game state to be evaluated, at tick T. $d_0(i)$ and $d_T(i)$ denote a measure of the distance to the object of type i that is closest to the avatar, in the root state s_0 or the state s_T, respectively. Intuitively, this means that the evaluation function rewards moving closer towards the closest object of a type i if it has a positive weight w_i, or rewards moving away from the closest object of a type i if it has a negative weight w_i. Note that only the *closest* object of every type i has an influence on the evaluation function, and other objects of the same type are not taken into account. The main reason for this is to avoid the avatar getting stuck exactly halfway between two objects of the same type i with a positive weight w_i, but it also reduces the computational cost of computing distances. The method used for determining these distances is described in more detail after the discussion about the weights.

This evaluation function is used as follows. Let $X(s_0)$ denote the evaluation of the current game state s_0, and let $X(s_T)$ denote the evaluation of the final state s_T of an MCTS simulation, both using the standard evaluation function of Equation 3.1. In cases where $X(s_0) = X(s_T)$, it is desirable to use the new evaluation function in an attempt to differentiate s_T from other states at the end of other simulations that also have the same standard evaluation. Therefore, $Eval_{KB}(s_T)$ is computed, normalized to lie in $[0, 0.5]$ based on all previously generated $Eval_{KB}(s)$ values, and added to $X(s_T)$. The reason for normalizing the evaluation to the interval $[0, 0.5]$ is that all the games available so far in the GVG-AI framework do not ever have any changes in score ΔX with $0 < \Delta X < 1$. This means that, by normalizing $Eval_{KB}(s)$ to $[0, 0.5]$, the addition of the heuristic evaluation for one game state will always be considered to be less important than a real change in score observed in another game state.

In cases where $X(s_0) \neq X(s_T)$, KBE is *not* used to change the evaluation $X(s_T)$. The main reason for this is simply because there already is an observed change in score that can be used to guide future MCTS simulations, so it is not considered to be necessary to compute $Eval_{KB}(s_T)$. Therefore, the computational cost of computing this evaluation can be saved. A different variant, where KBE is used even if $X(s_0) \neq X(s_T)$, has also been implemented and is tested in Chapter 5. This variant is only used when it is explicitly stated in this thesis.

Determining the Weights

As described above, a weight w_i is intended to have a high value for a type i if it is good for the avatar to move towards objects of that type, and a low value if it is bad for the avatar to move towards such objects. Because the rules of the game are not provided to the agent in GVGP, these values should be based on experience gained from, for instance, MCTS simulations. In a similar way as described by Perez *et al.* (2014), Van Eeden (2015), the experience that the values of the weights are based on consists of collision events observed in MCTS simulations and simultaneously observed changes in game score.

Let $E(s_t)$ denote the set of collision events that occurred during the transition from a state s_{t-1} to a state s_t, where s_t is a state generated in an MCTS simulation (and s_{t-1} is either the root state or also a state generated in the same simulation). This set can be obtained by retrieving a list of all collision events that have been observed since the start of the game in s_t, and removing the first n of them, where n is the size of the equivalent list for s_{t-1}. The events $(o_1, o_2) \in E(s_t)$ can be described as pairs of objects that collided, where o_1 is either the avatar or an object created by the avatar (such as a sword, or a missile), and o_2 is some other object (such as a wall, or an alien).

Let $\Delta_t = X(s_t) - X(s_{t-1})$ denote the difference in evaluation going from s_{t-1} to s_t. The idea is to give every event (o_1, o_2) "credit" for the evaluation difference Δ_t. Let i denote the type of the object o_2. Then, if events with objects of type i are frequently observed together with a positive change in evaluation $\Delta_t > 0$, it is assumed to be beneficial for the avatar to move towards objects of type i. Similarly, if events with objects of type i are frequently observed together with a negative change in evaluation $\Delta_t < 0$, it is assumed to be detrimental for the avatar to move towards objects of type i. Finally, if events with objects of type i are frequently observed together with no change in evaluation ($\Delta_t = 0$), objects of that type are assumed to be irrelevant.

More formally, for every state s_t generated in an MCTS simulation, knowledge is extracted from the experience $E(s_t)$ and Δ_t as follows. For every object type i, the following two values are stored; Δ_i, which is the sum of all changes in evaluations Δ_t that are observed at the same time as a collision event with an

Table 4.1: Initial Values of Weights

Object Category	**Initial w_i Value**
Static	0
NPC	0.1
Movable	0.25
Resource	1
Portal	1

object of type i, and n_i, which is a counter to keep track of how often such a collision event is observed. In other words, for every event $(o_1, o_2) \in E(s_t)$, where o_2 is an object of the type i, Δ_t is added to Δ_i, and n_i is increased by 1. Δ_i and n_i are both initialized to 0 whenever an object of type i is observed for the first time in a game. Δ_i and n_i can then be used to compute the average change in score $\overline{\Delta}_i = \frac{\Delta_i}{n_i}$ for collision events for every object type i.

The weight w_i is always updated immediately after updating Δ_i and n. This is done using the update rule given by Formula 4.3.

$$w_i \leftarrow w_i + (\overline{\Delta}_i - w_i) \times \alpha_i \tag{4.3}$$

α_i is a learning rate that is initialized to 0.8 for every type, and updated as given by Formula 4.4 after updating w_i.

$$\alpha_i \leftarrow \max(0.1, 0.75 \times \alpha_i) \tag{4.4}$$

These update rules essentially come down to using a technique like gradient descent to minimize the function $f(w_i) = |\overline{\Delta}_i - w_i|$, which is trivially minimized by setting w_i to $w_i = \overline{\Delta}_i$. The main reason for not doing this directly is to avoid relying too much on information gained from only a small number of observations.

The agent is motivated to explore the effects of collision events that have been observed rarely (or never) in two ways. First, at the start of every game tick, all weights w_i are increased by 10^{-4}. This keeps the agent interested in re-visiting object types for which the weights previously (nearly) converged to 0. Secondly, most weights are initialized to positive values. This initialization rewards the agent for exploring early in the game. The exact values that weights are initialized to are given in Table 4.1. These values have been chosen manually. Even though it is more common in the games of the GVG-AI framework for collisions with NPCs to be detrimental than beneficial, those weights are still initialized to a small positive value to ensure early exploration. Static objects have an initial weight of 0 because their purpose is typically only to block movement, but exploration of collision events with those objects will still be motivated over time due to the automatic increase of all weight values every tick.

Whenever a weight w_i gets a value $w_i < 10^{-4}$, the corresponding type i is ignored in the evaluation function given by Equation 4.2. If included, such a type would have a very small influence on the final value of the function, meaning that it is often safe to ignore it. Excluding such a type from the equation means that $d_T(i)$ no longer needs to be computed. That computation, which is described in detail below, can take a significant amount of time.

Computation of Distances

The evaluation function of Equation 4.2 requires, for all object types i with a weight w_i that is considered to be relevant ($w_i \geq 10^{-4}$), two measures of distance; $d_0(i)$ and $d_T(i)$. Perez *et al.* (2014) used Euclidean distances for similar variables in their evaluation function. They noted that, in some games, this results in the avatar getting stuck due to not taking into account the presence of movement-blocking obstacles, and suggested that the use of a pathfinding algorithm such as A* (Hart, Nilsson, and Raphael, 1968) may be beneficial. One of the extensions tested by Van Eeden (2015) was to replace these Euclidean distances with distances computed using A*.

In this thesis, the A* algorithm is also used to compute all the required distances $d_0(i)$ and $d_T(i)$. For all types i, the distances $d_0(i)$ are computed once for the root state at the start of every tick. This is also done for types with an irrelevant weight $w_i < 10^{-4}$, because the weight may change later in the same tick due to new knowledge gained from simulations, and become relevant. The distances $d_T(i)$ are computed at the end of every simulation where the heuristic evaluation function is used (i.e. simulations that ended in a state s_T with $X(s_0) = X(s_T)$) only for object types i with a relevant weight $w_i \geq 10^{-4}$. The implementation, described in more detail below, uses some optimizations previously described by Van Eeden (2015), and some other optimizations.

Let s_0 denote the root state, for which all distances $d_0(i)$ have already been computed at the start of the current game tick as described above. Let s_T denote the final state of a simulation, for which some distances $d_T(i)$ need to be computed. As described earlier, $d_T(i)$ is defined as the distance between the avatar and the *closest* object of type i in s_T. Under the assumption that an A* algorithm is available that can compute the distance between two objects, a naive algorithm to compute the distances $d_T(i)$ is given by Algorithm 4.2. This implementation is referred to as naive because it simply computes the exact distance for every object of every relevant type i, and memorizes the lowest value for every type.

Algorithm 4.2 Computing the $d_T(i)$ values (naive implementation)

Require: Game state s_T
1: **function** ComputeDistances
2: **for** every object type i with relevant weight $w_i \geq 10^{-4}$ **do**
3: $d_T(i) \leftarrow mapWidth \times mapHeight$
4: **for** every object $o \in s_T$ with type i **do**
5: $d_T(i) \leftarrow \min(d_T(i), \text{A*}(o, avatar))$
6: **end for**
7: **end for**
8: **end function**

This can be optimized using the fact that it is not necessary to compute the exact distance for every object of every type. It is only necessary to compute the exact distance for the *closest* object, and prove that all other objects of the same type have a greater distance (where the exact value of that greater distance does not matter). A small modification of the A* algorithm (described in more detail below) allows for an earlier termination when it is known that it cannot improve upon a previously found shortest distance. The idea is to sort all objects of a given type based on an estimate of the distance that is cheap to compute (such as the Manhattan distance) before looping through the objects. Finding a short distance for an object near the front of this sorted list allows subsequent A* searches to terminate early (sometimes instantly) for objects closer to the end of the list. Pseudocode of this implementation can be found in Algorithm 4.3. Using profiling software, this optimization was found to noticeably reduce the amount of time spent pathfinding in games with many objects, such as *Boulderdash*.

Algorithm 4.3 Computing the $d_T(i)$ values (optimized implementation)

Require: Game state s_T
1: **function** ComputeDistances
2: **for** every object type i with relevant weight $w_i \geq 10^{-4}$ **do**
3: $d_T(i) \leftarrow mapWidth \times mapHeight$
4: $O \leftarrow$ list of all objects $o \in s_T$ with type i
5: sort O in increasing order of Manhattan distance to $avatar$
6: **for** every $o \in O$ **do**
7: $d_T(i) \leftarrow \min(d_T(i), \text{A*}(o, avatar, d_T(i)))$
8: **end for**
9: **end for**
10: **end function**

As also described by Van Eeden (2015), A* has been implemented to take objects of the "wall" type into account as obstacle. This is an object type that is defined for all games in the GVG-AI framework. In almost all games, it is impossible for existing wall objects to disappear or for new objects to appear, and in almost all games they block movement. Many games also have different object types that can block movement, or objects that cause the avatar to teleport to a different location upon collision. Taking these objects into account for pathfinding would require comparing game states before and after playing movement actions to interpret the influence of those objects on movement. This has not been implemented for this thesis, but it could be beneficial to do so in future work.

A* has been implemented to interpret the game map as a 4-connected grid, meaning that the avatar is assumed not to be capable of diagonal movement. This assumption is made because the only movement actions available in GVG-AI are *Left*, *Up*, *Right*, and *Down*. However, in some games it may be inaccurate because the avatar could have "momentum" from previously played actions and therefore not only move in the direction corresponding to the last played action. Van Eeden (2015) does not explicitly mention also using a 4-connected grid, but the source code of the AIJim agent by the same author shows that it works in this way. The implementation also assumes that every movement to a neighbour has the same cost of 1.

Algorithm 4.4 shows pseudocode of the A* algorithm (Hart *et al.*, 1968) for computing the distance $d(o, avatar)$ between an object o (located at (x_o, y_o)), and the *avatar* (located at (x_a, y_a)). It is a Best-First Search algorithm. It stores cells to be processed in a collection referred to as the *openSet*, starting with only

Algorithm 4.4 A* for computing distances

```
1:  function A*(o, avatar, d_T(i))
2:      if (x_o, y_o) and (x_a, y_a) not connected then
3:          return mapWidth × mapHeight
4:      end if

5:      openSet ← ∅
6:      closedSet ← ∅
7:      openSet.Add((x_a, y_a))
8:      g(x_a, y_a) ← 0
9:      f(x_a, y_a) ← ManhattanDistance((x_a, y_a), (x_o, y_o))

10:     while openSet ≠ ∅ do
11:         (x, y) ← remove (x, y) that minimizes f(x, y) from openSet

12:         if x = x_o and y = y_o then
13:             return f(x, y)
14:         end if
15:         if f(x, y) ≥ d_T(i) then
16:             return f(x, y)
17:         end if

18:         closedSet.Add((x, y))
19:         for every neighbour (x', y') of (x, y) do
20:             if (x', y') ∉ closedSet then
21:                 openSet.Add((x', y'))
22:                 g(x', y') ← g(x, y) + 1
23:                 f(x', y') ← g(x', y') + ManhattanDistance((x', y'), (x_o, y_o))
24:             end if
25:         end for
26:     end while
27: end function
```

(x_a, y_a). For every cell (x, y), a lower bound on the cost of moving from (x_a, y_a) through (x, y) to (x_o, y_o) is defined as $f(x, y) = g(x, y) + h(x, y)$, where $g(x, y)$ is the cost of moving from (x_a, y_a) to (x, y), and $h(x, y)$ is an admissible heuristic estimate for the cost of moving from (x, y) to (x_o, y_o). The AIJim agent uses the Euclidean distance for $h(x, y)$. In this thesis, the Manhattan distance is used instead because it is cheaper to compute, often more accurate than the Euclidean distance on a 4-connected grid, and still admissible on a 4-connected grid. The algorithm prioritizes processing cells (x, y) with a low cost $f(x, y)$. Processing (x, y) consists of checking if it is the goal cell (x_o, y_o) and terminating if it is, and otherwise generating the neighbours (x', y') of (x, y) and adding them to the *openSet*. When cells are processed, they are marked by placing them in a *closedSet*, which prevents them from being processed again later. It is not necessary to re-visit these cells because the Manhattan distance is a "consistent" heuristic (Hart *et al.*, 1968).

Lines 2-4 of Algorithm 4.4 are an optimization also described in (Van Eeden, 2015). In the 1 second of processing time available for initialization at the start of every game, the map is analyzed to store for every pair of cells whether or not they are connected. In this case, cells are referred to as being connected if and only if it is possible to move from one to the other, without walking through obstacles (they do not necessarily have to be direct neighbours). This can be stored in $O(w \times h)$ memory, where w and h denote the width and the height of the map, respectively, by assigning a "colour" (simply represented by an integer) to every cell, such that all connected cells have the same colour. When an object o and the *avatar* are known not to be in connected cells, an upper bound on the distance can immediately be returned without performing any search. This upper bound, $w \times h$, is the length of a path that visits every cell once.

The *openSet* of the algorithm is often implemented as a priority queue, such as a binary heap, which supports the removals and insertions of lines 11 and 21, respectively, in $O(log(n))$ time. The implementation of this thesis uses a combination of a few different collections for the *openSet*. The highest priority collection (from which elements are removed first whenever it is not empty) is simply a stack, to which a neighbour (x', y') is pushed whenever it has a total cost equal to its parent's cost ($f(x', y') = f(x, y)$). The reasoning behind this is as follows. The cost $f(x', y')$ of a neighbour can never be smaller than the cost $f(x, y)$ of the parent. Because (x, y) is currently being processed, none of the other cells currently in the *openSet* can have a cost greater than $f(x, y)$. Therefore, a neighbour with a cost $f(x', y') = f(x, y)$ does not need to be inserted with a lower priority than any other cells in the *openSet*, but can automatically be given an equal priority to any other cells in the same stack (or a higher priority than all other cells in the *openSet* if this specific stack is empty). Pushing to and popping from this stack can be done in (amortized) constant time. This optimization is referred to as A* variant 1 (Sun *et al.*, 2009).

Additionally, because every movement to a neighbour is assumed to have a discrete cost of 1, the *openSet* can be implemented using multiple stacks, where a stack i contains cells (x, y) with a cost $f(x, y) = |x_o - x_a| + |y_o - y_a| + i$, where $|x_o - x_a| + |y_o - y_a|$ denotes the Manhattan distance from o to the avatar (which is the minimum cost of any cell). For this thesis, the algorithm has been implemented to use 4 such stacks (for $i \in [1, 4]$), and fall back to a binary heap for any cell with $i > 4$. It is likely possible to use a more efficient implementation, where an extra offset is used to "shift" elements in stacks with higher values for i to stacks with lower values for i once a number of stacks with low i have run empty. This could potentially entirely eliminate the need of a binary heap, but this has not been implemented and tested yet.

Lines 15-17 implement the optimization that is described above, where A* terminates early when the exact distance has not yet been computed, but it has been proven to be at least as large as the best distance $d_T(i)$ found so far. This is especially useful on maps with few obstacles. For instance, if there are no obstacles between the avatar and the closest object o of a type i, every call to A* except for the first call in Algorithm 4.3 terminates after only a single iteration.

Comparison to Previous Research

The main difference between KBE as described here, and the implementation described by Perez *et al.* (2014) is in the way that distances are computed. This was also done by Van Eeden (2015), but using fewer optimizations as described above (for instance, A* variant 1 was not used). However, the evaluation function used in this thesis (Equation 4.2) is also different from the ones of Perez *et al.* (2014) and Van Eeden (2015).

Perez *et al.* (2014) add the quantity given by Equation 4.5 to the evaluation of any terminal state s_T where $X(s_0) = X(s_T)$.

$$Reward = \beta \times \Delta D + \alpha \times \Delta Z \tag{4.5}$$

In this equation, α and β are parameters set to $\alpha = 0.66$ and $\beta = 0.33$. ΔD is defined in a way to have a similar effect as the evaluation function used in this thesis, given by Equation 4.2. ΔD is intended to reward moving towards objects that appear to be beneficial according to previously obtained experience, or objects of types for which no collision events have been observed yet. ΔZ is a quantity that rewards simulations where events are observed with object types that have not yet been observed frequently. This is intended to motivate exploration to gain new knowledge, but is not effective because it is only a retroactive reward given to a simulation in which the new knowledge has already been obtained.

Van Eeden (2015) proposes a change where exploration is instead motivated by rewarding movement towards object types for which collision events have been observed relatively rarely in comparison to other object types. This is more effective at rewarding exploration. It consistently tries to keep a balance between exploitation and exploration.

In this thesis, exploration is only motivated by using a positive initial value for the weights w_i, and by slowly increasing the value of all w_i over time. The initial weight values result in proactive motivation to explore object types with a low number of observed events (but not necessarily only types with 0 observations). The slow increase of all weight values over time results in a motivation to re-visit object types for which the weight previously converged to values close to 0. This means that the approach of this thesis likely has more exploration than the approach of Perez *et al.* (2014), but less exploration than the approach of Van Eeden (2015).

4.2.5 Deterministic Game Detection (DGD)

Background

Many of the highest-ranking agents in the GVGP competitions of 2015 attempted to detect when the game being played was deterministic, and adjust the algorithms used according to this information. This idea is referred to in this thesis as *Deterministic Game Detection* (DGD). Table 4.2 lists some of these agents with their rankings in the competitions, and describes how they responded to the detection of deterministic games. To the best of the authors' knowledge, there are no publications available that describe in detail how these

Table 4.2: Deterministic Game Detection in GVGP Competitions of 2015

Agent Name	Rankings	Response to Deterministic Game Detection
RETURN42	1^{st} CIG, 4^{th} CEEC, 4^{th} GECCO	A*-based algorithm in deterministic games, Genetic Algorithm otherwise
YOLOBOT	2^{nd} CIG, 3^{rd} CEEC, 1^{st} GECCO	Exhaustive search in deterministic games, MCTS otherwise
THORBJRN	12^{th} CIG, 9^{th} CEEC, 2^{nd} GECCO	Breadth-First Search in deterministic games, MCTS otherwise

agents (or any other GVGP agents) detect whether or not a game is deterministic. However, the source code is publicly available, which reveals that they all use a similar implementation. The implementation for this thesis, which is based on the same ideas, is described next.

Game Classification

Let s_0 denote the initial game state of a game. In the 1 second of initialization time that is available at the start of every game, M sequences of actions are randomly generated, with a length of N actions per sequence. All M action sequences are used to advance copies of s_0 by up to N ticks (possibly fewer if a terminal state is reached before playing N actions). This process is repeated R times per action sequence.

For all M action sequences, there is a set of R possible states that playing that action sequence can lead to. If, for all M action sequence, no difference can be found between the R final states for that sequence, the game is classified as deterministic. Otherwise, the game is classified as nondeterministic. Additionally, if in any game state generated in this process it is detected that there are NPCs in the game, it is immediately classified as a nondeterministic game. This is done because almost all NPCs can have random behaviour. In rare cases, this procedure can result in an incorrect classification, but that does not happen frequently.

If a game is classified as nondeterministic, DGD does not have any effect (except for the computational overhead for classification at the start of the game). Otherwise, it is used to modify the selection step of MCTS, and the Tree Reuse and Novelty-Based Pruning enhancements. This is different from the agents listed in Table 4.2, which switch to entirely different algorithms.

MCTS Selection in Deterministic Games

In deterministic games, as soon as one MCTS simulation ends in a game state s_T with a promising evaluation $X(s_T)$, it is known for sure that this evaluation really can be obtained; there is no opposing player that can deny the agent from reaching that state, and the probability that something unexpected happens when playing the same action sequence is 0. Therefore, it is reasonable to assume that the maximum score $X_{max}(n)$ that was ever backpropagated through a node n can be used to guide future MCTS simulations.

This is done by modifying the formula used in the selection step of MCTS; Equation 3.3 by default, or Equation 4.1 if Progressive History is used. The $\overline{Q}(s_i)$ term at the start of these equations is replaced by $(1-\alpha_{DGD})\times\overline{Q}(s_i)+\alpha_{DGD}\times Q_{max}(s_i)$, with $\alpha_{DGD} \in [0,1]$. $Q_{max}(s_i)$ is the $X_{max}(s_i)$ value normalized. This idea of mixing the average evaluation with the maximum evaluation, referred to as *MixMax*, was previously used in Super Mario Bros (Jacobsen, Greve, and Togelius, 2014). It has also previously been tested in GVGP by Frydenberg *et al.* (2015), but they used it in all games (not only deterministic games). This thesis uses $\alpha_{DGD} = 0.25$.

Novelty-Based Pruning and Tree Reuse in Deterministic Games

In Subsection 4.2.3, it is described that the results of novelty tests in the first ply below the new root are reset when Novelty-Based Pruning (NBP) is combined with Tree Reuse (TR). This is done to reduce the effect that nondeterminism in games can have on these novelty tests. Additionally, it also describes that the weight of results based on old simulations is reduced by multiplying them with a factor γ, because some of these simulations may be irrelevant due to nondeterminism.

When a game is classified as deterministic, these cases no longer need to be considered. Therefore, when DGD is combined with TR, the parameter γ is always set to 1 in deterministic games. When NBP is also included, the results of old novelty tests are no longer reset when reusing a previous tree in a deterministic game.

4.2.6 Temporal-Difference Tree Search (TDTS)

Temporal-Difference Backups

Temporal-Difference Tree Search (Vodopivec, 2016) (TDTS) is a generalization of MCTS that uses *temporal-difference (TD) learning* (Sutton, 1988) to estimate the values of states based on MCTS simulations, instead of the default Monte-Carlo (MC) backups. Using MC backups, the value of a state (or node) is simply estimated to be the average of the final evaluations of all simulations going through that state (or node). Using TD backups, the value of a state (or node) is instead estimated based on the estimated value of the successor in a simulation, and any immediate rewards for reaching that state (or node). In the field of Reinforcement Learning (Sutton and Barto, 1998), TD methods have been found to usually converge to optimal or good estimates more quickly than MC methods.

An example of a TDTS implementation that is proposed by Vodopivec (2016) is *Sarsa-UCT(λ)*. Sarsa-UCT(λ) uses the update rule of the Sarsa algorithm (Rummery and Niranjan, 1994) to update estimates of the values of nodes. Vodopivec (2016) also tested this algorithm in the GVGP competition of 2015, where

it outperformed a standard UCT implementation. Algorithm 4.5 shows pseudocode for the backpropagation step in a standard UCT implementation, and in Sarsa-UCT(λ). The pseudocode of Sarsa-UCT(λ) as written here assumes that the MCTS tree is expanded with one node for every state generated in the play-out step, as is done in the implementation for this thesis.

Algorithm 4.5 Backpropagation Step for UCT and for Sarsa-UCT(λ)

Require: Sequences of traversed nodes $\langle n_0, n_1, \ldots, n_T \rangle$ and states $\langle s_0, s_1, \ldots, s_T \rangle$

1: **function** UCT BACKPROPAGATE
2: **for** $t \leftarrow T, T-1, \ldots, 0$ **do**
3: $n_t.Visits \leftarrow n_t.Visits + 1$
4: $n_t.TotalScore \leftarrow n_t.TotalScore + X(s_T)$ ▷ Allows computing $\overline{X}(n_t) = \frac{n_t.TotalScore}{n_t.Visits}$
5: **end for**
6: **end function**

7: **function** SARSA-UCT(λ) BACKPROPAGATE ▷ Adapted from pseudocode in (Vodopivec, 2016)
8: $\delta_{sum} \leftarrow 0$
9: $V_{next} \leftarrow 0$
10: **for** $t \leftarrow T, T-1, \ldots, 1$ **do**
11: $R \leftarrow X(s_t) - X(s_{t-1})$ ▷ immediate reward
12: $V_{current} \leftarrow V(n_t)$ ▷ current estimate of value of n_t
13: $\delta \leftarrow R + \gamma V_{next} - V_{current}$ ▷ single-step TD error
14: $\delta_{sum} \leftarrow \lambda\gamma\delta_{sum} + \delta$ ▷ accumulated TD error decayed according to eligibility trace
15: $n_t.Visits \leftarrow n_t.Visits + 1$
16: $\alpha \leftarrow \frac{1}{n_t.Visits}$ ▷ this gives every simulation an equal weight when updating value
17: $V(n_t) \leftarrow V(n_t) + \alpha\delta_{sum}$
18: $V_{next} \leftarrow V_{current}$
19: **end for**
20: **end function**

The backpropagation step of Sarsa-UCT(λ) has two parameters; $\gamma \in [0, 1]$ and $\lambda \in [0, 1]$. γ discounts long-term rewards if $\gamma < 1$, meaning that a reward that can be obtained quickly is prioritized over an equally large reward that requires more time to obtain. In (Vodopivec, 2016), as well as this thesis, $\gamma = 1$ because the amount of time that it takes to obtain a score increase in GVGP rarely matters. λ is used to implement the concept of *eligibility traces* (Sutton and Barto, 1998). Intuitively, eligibility traces are used to update value estimates less quickly according to distant rewards than according to rewards that are nearby, because distant rewards are assumed to be less reliable. An important difference between γ and λ is that changing the value of γ changes the value estimates that the algorithm converges to given an infinite amount of processing time, whereas changing the value of λ can only affect the rate of convergence.

The pseudocode of Sarsa-UCT(λ) as described above takes into account at which point in time the agent receives a reward, because line 11 takes the difference between the evaluations of two consecutive game states. This means that nodes deep in a simulation are not rewarded for score gains obtained earlier in the same simulation. This is different from the MC backups of the standard UCT, which takes the difference between evaluations of the first and last states of a simulation, and rewards all nodes traversed in that simulation for any change in score. Vodopivec (2016) took intermediate rewards into account in other domains, but not in GVGP. In GVGP, the value of R was set to $R = X(s_T) - X(s_0)$ at the end of the simulation (i.e., the first iteration of the loop starting in line 10), and set to $R = 0$ for all other state transitions. In such an implementation, Sarsa-UCT(λ) no longer takes into account intermediate rewards (just like the standard UCT implementation). For this thesis, both variants have been implemented and tested.

When intermediate rewards are not taken into account, and $\lambda = 1$, the two backpropagation functions described above are only different in that the MC backups of UCT include the evaluation of the root state $X(s_0)$, whereas the TD backups of Sarsa-UCT(λ) do not. This is discussed in more detail below. When $\lambda < 1$, distant rewards are assumed to be less reliable than rewards that are nearby. In MCTS, this is a

valid assumption, because a distant reward is the result of a longer sequence of actions than a reward that is nearby. In MCTS, a longer sequence of actions means that more actions were chosen according to the selection and/or play-out policies of MCTS. Both of these can be considered to be "unreliable" policies. The selection policy includes intentional exploration of lines of play that are estimated to be worse than other alternatives, but have low visit counts, and the play-out policy chooses actions (semi-)randomly. This means that it is a reasonable assumption that distant rewards are less reliable than rewards that are nearby. TD backups with $\lambda < 1$ are expected to be less susceptible to estimating values incorrectly because of unreliable, distant backups than MC backups.

Selection in TDTS

Vodopivec (2016) changed the UCB1 selection policy to normalize and use the value estimate $V(n_t)$ of a node n_t obtained through TD backups, instead of the $\overline{X}(n_t)$ value obtained through MC backups. It is not straightforward to combine this approach with other enhancements for the selection step, such as *Progressive History* (Subsection 4.2.1) and *MixMax backups* (Subsection 4.2.5), because the $V(n_t)$ values of Sarsa-UCT(λ) have a different range than the $\overline{X}(n_t)$ values of UCT;

- $V(n_t)$ is an estimate of the direct reward for traversing from the previous node n_{t-1} to n_t, plus an estimate of the rewards that can be obtained in the tree below n_t.
- $\overline{X}(n_t)$ is an estimate of the score that has already been accumulated in the predecessor n_{t-1}, plus the same estimates described above for $V(n_t)$.

For example, let n_t denote the root node in a game of *Aliens* where a number of game ticks have already passed. Suppose that the agent has already accumulated a game score of 30 points. Let n_{t+1} denote a successor of n_t where the transition from n_t to n_{t+1} does not result in a change in game score. Then, $\overline{X}(n_{t+1}) = 30$, but $V(n_{t+1}) = 0$. Without any other changes, the extra values that PH and MixMax add to the formula of the selection policy are expected to be in a similar range as the $\overline{X}(n_t)$ values. Therefore, to combine Sarsa-UCT(λ) with PH and MixMax, either the extra values used by those enhancements should be scaled to the same range as $V(n_t)$, or the $V(n_t)$ values should be scaled to the same range as the $\overline{X}(n_t)$ values. In this thesis, before using $V(n_t)$ values to replace $\overline{X}(n_t)$ in the selection policy, they are scaled by adding the evaluation $X(s_{t-1})$ of the previous game state in that particular simulation to $V(n_t)$. In comparison to the original implementation of Sarsa-UCT(λ), this means that the value estimates of candidate successors that the selection policy can choose are likely closer together, and therefore it is expected that the exploration parameter C should have a lower value.

Vodopivec (2016) also describes *space-local value normalization* as a different technique for normalizing the estimates of values to $[0, 1]$ before using them in the selection step. This technique uses a smaller range of more local bounds, instead of the global bounds $\hat{X}_{min}$ and $\hat{X}_{max}$ described in Section 3.1. This technique was found to result in an improved performance by Vodopivec (2016). For this thesis, it has only been evaluated very briefly, but was not found to be beneficial and is not described further. This is likely because it may have a large impact on the optimal value for the C parameter, and also interacts with PH and MixMax.

Chapter 5

Experiments & Results

This chapter discusses experiments that have been done to evaluate the impact of the enhancements described in the previous chapter on the performance of MCTS in GVGP. The setup of the experiments is described first, and the results are described afterwards.

5.1 Setup

To evaluate the impact of the discussed enhancements on the performance of MCTS in GVGP, a number of experiments have been performed. The main measure of performance used is the average win percentage over sixty different games included in the GVG-AI framework, using the same amount of processing time as allowed in the competition (Perez *et al.*, 2016). For all win percentages listed in the results below, 95% confidence intervals are provided. Some enhancements are also evaluated using other measures.

First, a baseline MCTS agent without any of the discussed enhancements has been implemented. This implementation is based on the MaastCTS agent (Schuster, 2015), but contains a number of modifications. For instance, it expands more than one node per simulation, and it uses the 1 second of processing time available at the start of every game to run simulations. This agent is simply referred to as "MCTS". Next, the *Breadth-First Tree Initialization and Safety Prepruning* (BFTI) enhancement is evaluated by adding it individually to MCTS. All other enhancements are evaluated afterwards by adding them individually to an agent with BFTI enabled. The reason for enabling BFTI in the evaluation of the other enhancements is provided in the discussion of the results of BFTI. Finally, a number of experiments are discussed where multiple enhancements are combined. This also includes experiments to test variations of enhancements in a setting with multiple enhancements combined.

The games for all experiments were played using revision `24b11aea75722ab02954c326357949b97efb7789` of the GVG-AI framework (`https://github.com/EssexUniversityMCTS/gvgai`), running on a CentOS Linux server consisting of four AMD Twelve-Core OpteronT 6174 processors (2.20 GHz). This version of the framework contains a number of optimizations that, on average, significantly increase the speed of the *advance* and *copy* operations of the framework's forward model. These optimizations were implemented by the author of this thesis, in cooperation with Diego Perez-Liebana (one of the authors of the GVG-AI framework and organizers of the competitions). They were found to increase the average number of simulations per tick of the Sample Open-Loop MCTS controller of the GVG-AI framework by more than 50% on average (and more than 100% in some games). This should be taken into account when comparing the results of this thesis to those of publications before 2016. The hardware used for the experiments of this thesis is significantly slower than the hardware used in the competition (2.90 GHz) (Perez *et al.*, 2016). This means that any agent described in these experiments may have a higher average win percentage on the competition server.

5.2 Results

5.2.1 Benchmark Agents

In this experiment, each of the sixty games was played 100 times (with 20 repetitions per level) by four different benchmark agents. Three of these agents (SOLMCTS, IW(1), and YBCriber) were not implemented specifically for this thesis, but they have been included in the experiments to make sure that results are available using the same hardware and framework version. This allows for a fair comparison of the performance of these agents with agents developed for this thesis.

- **SOLMCTS**: The Sample Open-Loop MCTS controller that is included in the GVG-AI framework (Perez *et al.*, 2016).
- **MCTS**: The baseline MCTS agent developed for this thesis, without any enhancements. Based on MaastCTS (Schuster, 2015).
- **IW(1)**: The agent using IW(1), as described in (Geffner and Geffner, 2015).
- **YBCriber**: The agent that won the GVGP competition at the IEEE CEEC 2015 conference. It is based on Iterated Width, but also uses additional heuristics specific to GVGP. This agent is a good representation of the state of the art.

The important differences between the SOLMCTS and MCTS agents are listed in Table 5.1.

Table 5.1: Differences between the SOLMCTS and MCTS agents.

SOLMCTS	**MCTS**
Has C set to $C = \sqrt{2}$ for the UCB1 equation used in selection step.	Has C set to $C = 0.6$ for the UCB1 equation used in selection step.
Expands the search tree by only one node per simulation.	Expands the search tree by adding nodes for the entire play-out of every simulation.
Has a depth limit of 10 for the selection and play-out steps combined.	Has a depth limit of 10 for the play-out step alone.
Does not make use of the 1 second of initialization time available at the start of every game.	Makes use of the 1 second of initialization time available at the start of every game by running MCTS (essentially giving the first tick a significantly higher number of simulations).
Plays the action corresponding to the highest visit count at the end of every game tick.	Plays the action corresponding to the highest average score at the end of every game tick.

The win percentages of these four benchmark agents are shown in Table 5.2. It shows that, even with just the differences described above, the MCTS agent already has a significantly higher average performance than SOLMCTS. IW(1) can be said to outperform both SOLMCTS and MCTS on average with 95% confidence, but the performance gap between MCTS and IW(1) is small. YBCriber clearly has the best overall performance among these agents.

Table 5.2: Win Percentages of Benchmark Agents (100 runs per game / 20 runs per level)

	Win Percentage (%)			
Games	SOLMCTS	MCTS	IW(1)	YBCriber
aliens	100.0 ± 0.0	100.0 ± 0.0	96.0 ± 3.8	100.0 ± 0.0
bait	6.0 ± 4.7	5.0 ± 4.3	21.0 ± 8.0	55.0 ± 9.8
blacksmoke	1.0 ± 2.0	1.0 ± 2.0	0.0 ± 0.0	0.0 ± 0.0
boloadventures	0.0 ± 0.0	1.0 ± 2.0	0.0 ± 0.0	0.0 ± 0.0
boulderchase	13.0 ± 6.6	16.0 ± 7.2	18.0 ± 7.5	13.0 ± 6.6
boulderdash	6.0 ± 4.7	10.0 ± 5.9	3.0 ± 3.3	20.0 ± 7.8
brainman	7.0 ± 5.0	6.0 ± 4.7	0.0 ± 0.0	38.0 ± 9.5
butterflies	86.0 ± 6.8	99.0 ± 2.0	98.0 ± 2.7	100.0 ± 0.0
cakybaky	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
camelRace	15.0 ± 7.0	13.0 ± 6.6	10.0 ± 5.9	100.0 ± 0.0
catapults	0.0 ± 0.0	5.0 ± 4.3	6.0 ± 4.7	7.0 ± 5.0
chase	0.0 ± 0.0	3.0 ± 3.3	27.0 ± 8.7	45.0 ± 9.8
chipschallenge	11.0 ± 6.1	17.0 ± 7.4	4.0 ± 3.8	52.0 ± 9.8
chopper	14.0 ± 6.8	84.0 ± 7.2	2.0 ± 2.7	89.0 ± 6.1
cookmepasta	0.0 ± 0.0	1.0 ± 2.0	4.0 ± 3.8	19.0 ± 7.7
crossfire	4.0 ± 3.8	3.0 ± 3.3	27.0 ± 8.7	99.0 ± 2.0
defender	59.0 ± 9.6	69.0 ± 9.1	59.0 ± 9.6	84.0 ± 7.2
digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
eggomania	7.0 ± 5.0	15.0 ± 7.0	67.0 ± 9.2	82.0 ± 7.5
enemycitadel	2.0 ± 2.7	6.0 ± 4.7	0.0 ± 0.0	0.0 ± 0.0
escape	0.0 ± 0.0	0.0 ± 0.0	64.0 ± 9.4	100.0 ± 0.0
factorymanager	91.0 ± 5.6	88.0 ± 6.4	87.0 ± 6.6	100.0 ± 0.0
firecaster	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
firestorms	9.0 ± 5.6	4.0 ± 3.8	60.0 ± 9.6	100.0 ± 0.0
frogs	13.0 ± 6.6	10.0 ± 5.9	78.0 ± 8.1	80.0 ± 7.8
gymkhana	2.0 ± 2.7	0.0 ± 0.0	2.0 ± 2.7	1.0 ± 2.0
hungrybirds	37.0 ± 9.5	33.0 ± 9.2	57.0 ± 9.7	99.0 ± 2.0
iceandfire	0.0 ± 0.0	0.0 ± 0.0	20.0 ± 7.8	100.0 ± 0.0
infection	94.0 ± 4.7	96.0 ± 3.8	98.0 ± 2.7	100.0 ± 0.0
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	97.0 ± 3.3
jaws	78.0 ± 8.1	71.0 ± 8.9	44.0 ± 9.7	7.0 ± 5.0
labyrinth	11.0 ± 6.1	10.0 ± 5.9	40.0 ± 9.6	100.0 ± 0.0
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
missilecommand	58.0 ± 9.7	76.0 ± 8.4	79.0 ± 8.0	100.0 ± 0.0
modality	25.0 ± 8.5	25.0 ± 8.5	22.0 ± 8.1	80.0 ± 7.8
overload	13.0 ± 6.6	26.0 ± 8.6	54.0 ± 9.8	100.0 ± 0.0
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	3.0 ± 3.3
painter	63.0 ± 9.5	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
plants	7.0 ± 5.0	7.0 ± 5.0	3.0 ± 3.3	1.0 ± 2.0
plaqueattack	74.0 ± 8.6	96.0 ± 3.8	31.0 ± 9.1	100.0 ± 0.0
portals	13.0 ± 6.6	15.0 ± 7.0	62.0 ± 9.5	31.0 ± 9.1
racebet2	70.0 ± 9.0	62.0 ± 9.5	57.0 ± 9.7	100.0 ± 0.0
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
roguelike	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 5.0
seaquest	96.0 ± 3.8	84.0 ± 7.2	84.0 ± 7.2	70.0 ± 9.0
sheriff	99.0 ± 2.0	100.0 ± 0.0	87.0 ± 6.6	9.0 ± 5.6
sokoban	11.0 ± 6.1	24.0 ± 8.4	10.0 ± 5.9	69.0 ± 9.1
solarfox	0.0 ± 0.0	1.0 ± 2.0	0.0 ± 0.0	2.0 ± 2.7
superman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	63.0 ± 9.5
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
survivezombies	40.0 ± 9.6	45.0 ± 9.8	35.0 ± 9.3	43.0 ± 9.7
tercio	0.0 ± 0.0	1.0 ± 2.0	1.0 ± 2.0	28.0 ± 8.8
thecitadel	9.0 ± 5.6	14.0 ± 6.8	9.0 ± 5.6	57.0 ± 9.7
waitforbreakfast	19.0 ± 7.7	26.0 ± 8.6	0.0 ± 0.0	80.0 ± 7.8
whackamole	100.0 ± 0.0	97.0 ± 3.3	97.0 ± 3.3	95.0 ± 4.3
zelda	18.0 ± 7.5	35.0 ± 9.3	70.0 ± 9.0	100.0 ± 0.0
zenpuzzle	11.0 ± 6.1	62.0 ± 9.5	22.0 ± 8.1	21.0 ± 8.0
Total	26.5 ± 1.1	31.0 ± 1.2	33.6 ± 1.2	52.4 ± 1.3

5.2.2 Breadth-First Tree Initialization and Safety Prepruning

The *Breadth-First Initialization and Safety Prepruning* (BFTI) enhancement has been implemented and tested by adding it to the standard MCTS agent. Table 5.3 compares this new agent (referred to as BFTI) to the MCTS agent in terms of average win percentage and average number of simulations per game tick. The results of BFTI were obtained by playing 75 runs per game (15 runs per level). The results for MCTS were obtained in the experiment described above (100 runs per game, 20 runs per level). The M parameter, which determines the number of states that are generated for every action available in the root state to deal with nondeterminism, is set to $M = 3$. This value has been chosen manually. Geffner and Geffner (2015) used a very different value ($M = 10$) for a similar parameter for safety prepruning in IW. This difference is because, for IW, safety prepruning is the *only* method for dealing with nondeterminism. Open-loop MCTS also continues to take nondeterminism into account after the BFTI process, and can therefore afford to use a lower value for M.

Overall, the addition of BFTI appears to slightly decrease the average win percentage from 31.0% to 30.0%, but the 95% confidence intervals overlap. The most notable decrease in win percentage is in the game of *Chopper*, where there also is a significant decrease in the average number of simulations per tick. There are significant increases in win percentage in the games of *Firestorms* and *Frogs*. These are both games where there are many objects on the map that cause a loss upon collision with the avatar. This means that the safety prepruning of BFTI can frequently reduce the number of actions to be considered in the root node.

Figure 5.1 depicts, for the BFTI and MCTS agents, 95% confidence intervals for three statistics. The first bar for each agent is the percentage out of all games that was won. These are the same values as found in Table 5.3. The second bar for each agent is the percentage out of all games that was lost before 2000 ticks passed. The third bar for each agent is the percentage out of all games that was lost after 2000 ticks passed (where 2000 ticks is the maximum allowed duration for any game). In most games, losing a game before 2000 ticks pass means that the agent makes a clear mistake, such as colliding with a harmful object. Losing a game after 2000 ticks pass means that the agent failed to find a way to win, but at least managed to survive. Losing after 2000 ticks can be considered to be better than losing earlier. One reason for this is that surviving for a longer amount of time may enable the other enhancements discussed in this thesis to find more winning lines of play. For this reason, BFTI is included in the following experiments where these other enhancements are evaluated, even though it appears to slightly reduce the overall win percentage. Another reason why losses after 2000 ticks can be considered to be better than earlier losses, is that, in many games, early losses also cause a decrease in game score. Because the game score is used as a tie-breaker in the competition, BFTI could improve the competition ranking in games where it significantly increases the number of runs where the agent survives for 2000 ticks.

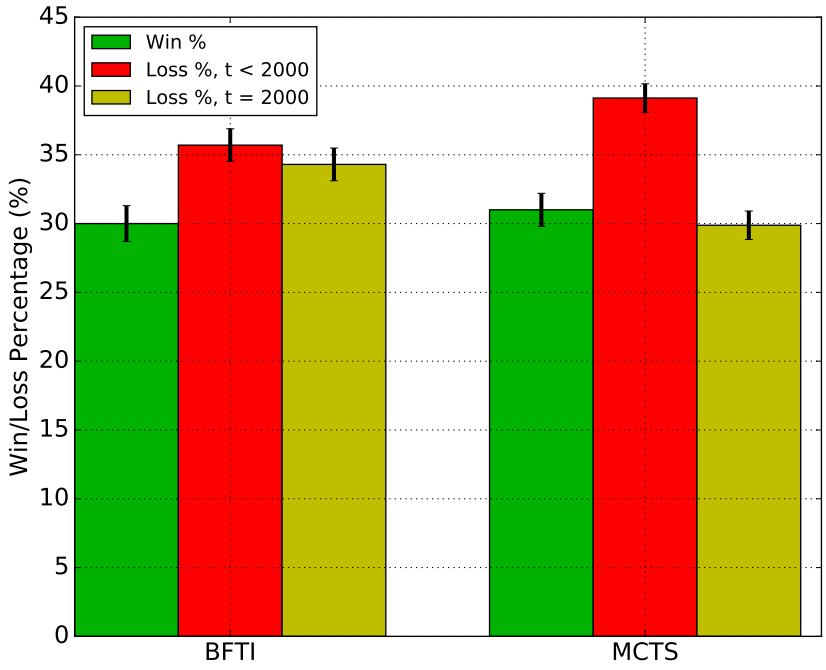


Figure 5.1: Comparison of BFTI and MCTS in terms of average win percentage, average loss percentage after less than 2000 ticks, and average loss percentage after 2000 ticks.

Table 5.3: Comparison of BFTI and MCTS in terms of Win Percentages and Avg. Simulations per Tick

	Win Percentage (%)		Avg. Simulations per Tick	
Games	BFTI	MCTS	BFTI	MCTS
aliens	100.0 ± 0.0	100.0 ± 0.0	45.5 ± 1.2	45.6 ± 1.5
bait	5.3 ± 5.1	5.0 ± 4.3	20.0 ± 2.8	20.7 ± 2.6
blacksmoke	0.0 ± 0.0	1.0 ± 2.0	9.5 ± 0.4	14.5 ± 0.5
boloadventures	0.0 ± 0.0	1.0 ± 2.0	2.6 ± 0.8	9.8 ± 0.5
boulderchase	13.3 ± 7.7	16.0 ± 7.2	12.2 ± 0.7	14.4 ± 0.5
boulderdash	9.3 ± 6.6	10.0 ± 5.9	8.7 ± 1.0	11.9 ± 0.5
brainman	2.7 ± 3.6	6.0 ± 4.7	22.4 ± 3.5	25.7 ± 3.3
butterflies	98.7 ± 2.6	99.0 ± 2.0	60.2 ± 1.0	59.6 ± 1.1
cakybaky	0.0 ± 0.0	0.0 ± 0.0	19.1 ± 1.6	23.9 ± 1.5
camelRace	17.3 ± 8.6	13.0 ± 6.6	43.7 ± 2.3	45.0 ± 3.0
catapults	4.0 ± 4.4	5.0 ± 4.3	45.4 ± 3.9	54.8 ± 4.6
chase	5.3 ± 5.1	3.0 ± 3.3	30.7 ± 1.0	30.1 ± 0.8
chipschallenge	17.3 ± 8.6	17.0 ± 7.4	19.3 ± 2.2	23.8 ± 2.0
chopper	36.0 ± 10.9	84.0 ± 7.2	5.2 ± 1.3	13.3 ± 0.8
cookmepasta	0.0 ± 0.0	1.0 ± 2.0	26.4 ± 3.3	25.6 ± 3.5
crossfire	1.3 ± 2.6	3.0 ± 3.3	26.0 ± 0.6	28.8 ± 0.6
defender	70.7 ± 10.3	69.0 ± 9.1	30.1 ± 1.7	30.4 ± 2.0
digdug	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.4	12.1 ± 0.7
eggomania	6.7 ± 5.6	15.0 ± 7.0	57.9 ± 1.1	72.5 ± 1.6
enemycitadel	1.3 ± 2.6	6.0 ± 4.7	4.6 ± 1.0	10.6 ± 0.5
escape	0.0 ± 0.0	0.0 ± 0.0	34.5 ± 1.1	36.5 ± 1.2
factorymanager	88.0 ± 7.4	88.0 ± 6.4	12.1 ± 1.2	15.2 ± 0.9
firecaster	0.0 ± 0.0	0.0 ± 0.0	26.9 ± 2.6	28.6 ± 2.4
firestorms	21.3 ± 9.3	4.0 ± 3.8	25.7 ± 0.4	29.5 ± 0.5
frogs	44.0 ± 11.2	10.0 ± 5.9	17.5 ± 2.4	39.3 ± 1.7
gymkhana	1.3 ± 2.6	0.0 ± 0.0	26.9 ± 1.0	32.3 ± 1.3
hungrybirds	34.7 ± 10.8	33.0 ± 9.2	35.0 ± 3.0	35.5 ± 3.0
iceandfire	0.0 ± 0.0	0.0 ± 0.0	53.5 ± 0.5	63.1 ± 0.7
infection	97.3 ± 3.6	96.0 ± 3.8	32.8 ± 0.6	33.5 ± 0.7
intersection	100.0 ± 0.0	100.0 ± 0.0	36.9 ± 1.1	42.2 ± 1.2
jaws	77.3 ± 9.5	71.0 ± 8.9	50.1 ± 4.9	47.0 ± 5.7
labyrinth	9.3 ± 6.6	10.0 ± 5.9	54.0 ± 0.8	56.4 ± 0.7
lasers	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 0.7	10.7 ± 0.3
lasers2	0.0 ± 0.0	0.0 ± 0.0	2.5 ± 1.0	9.8 ± 0.6
lemmings	0.0 ± 0.0	0.0 ± 0.0	53.6 ± 0.8	62.3 ± 0.9
missilecommand	69.3 ± 10.4	76.0 ± 8.4	139.1 ± 3.3	148.9 ± 4.7
modality	25.3 ± 9.8	25.0 ± 8.5	37.9 ± 15.1	67.1 ± 24.0
overload	17.3 ± 8.6	26.0 ± 8.6	13.5 ± 3.4	20.3 ± 2.6
pacman	0.0 ± 0.0	0.0 ± 0.0	8.5 ± 0.4	14.0 ± 0.3
painter	100.0 ± 0.0	100.0 ± 0.0	51.0 ± 7.1	81.2 ± 10.3
plants	4.0 ± 4.4	7.0 ± 5.0	29.5 ± 0.8	31.1 ± 0.8
plaqueattack	94.7 ± 5.1	96.0 ± 3.8	26.3 ± 1.1	27.5 ± 1.2
portals	13.3 ± 7.7	15.0 ± 7.0	32.5 ± 0.7	33.6 ± 1.1
racebet2	68.0 ± 10.6	62.0 ± 9.5	20.7 ± 0.5	24.3 ± 0.7
realportals	0.0 ± 0.0	0.0 ± 0.0	7.1 ± 2.2	15.1 ± 1.5
realsokoban	0.0 ± 0.0	0.0 ± 0.0	23.6 ± 1.9	29.2 ± 2.2
roguelike	0.0 ± 0.0	0.0 ± 0.0	22.5 ± 0.9	24.2 ± 0.8
seaquest	70.7 ± 10.3	84.0 ± 7.2	28.0 ± 2.0	31.6 ± 2.4
sheriff	100.0 ± 0.0	100.0 ± 0.0	31.9 ± 1.2	35.8 ± 1.4
sokoban	20.0 ± 9.1	24.0 ± 8.4	29.5 ± 2.4	28.7 ± 2.3
solarfox	1.3 ± 2.6	1.0 ± 2.0	74.2 ± 3.2	80.4 ± 3.7
superman	0.0 ± 0.0	0.0 ± 0.0	41.4 ± 0.9	47.8 ± 1.2
surround	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	105.9 ± 5.5
survivezombies	41.3 ± 11.1	45.0 ± 9.8	26.4 ± 0.8	27.9 ± 0.8
tercio	0.0 ± 0.0	1.0 ± 2.0	4.3 ± 1.7	10.1 ± 1.6
thecitadel	5.3 ± 5.1	14.0 ± 6.8	7.0 ± 1.6	15.5 ± 1.1
waitforbreakfast	20.0 ± 9.1	26.0 ± 8.6	35.8 ± 2.4	42.0 ± 2.7
whackamole	100.0 ± 0.0	97.0 ± 3.3	98.0 ± 2.4	118.3 ± 3.2
zelda	32.0 ± 10.6	35.0 ± 9.3	25.7 ± 1.6	27.9 ± 1.7
zenpuzzle	56.0 ± 11.2	62.0 ± 9.5	83.8 ± 7.6	94.6 ± 14.3
Total	30.0 ± 1.3	31.0 ± 1.2	31.0 ± 6.4	38.3 ± 7.3

5.2.3 Loss Avoidance

Table 5.4 shows the results obtained by adding *Loss Avoidance* (LA) to the BFTI agent. On average, LA gives a small, but statistically significant increase in win percentage from 30.0% to 33.3%. The win percentage is especially increased in games where the avatar is killed upon collision with certain static objects, such as *Catapults* and *Overload*. The large decrease in the average simulations per tick in these games also shows that the LA process is executed frequently in these games.

LA has a very detrimental impact on the win percentage in the game of *Jaws*. In this game, there are certain locations that have a low probability of spawning harmful objects. When a simulation ends with the avatar getting killed in such a position, the LA process re-generates states for the nodes traversed in that simulation. Because these locations only have a low probability of spawning harmful objects, it is likely that in such a second sequence of generating states by LA, a harmful object is no longer spawned. Therefore, LA causes the agent to have a tendency of ignoring the fact that these locations are dangerous to move to. One way to address this issue in future work could be to use *influence maps* (Millington and Funge, 2009) to keep track of how frequently losses are observed in certain cells of the map, and incorporate that information in a heuristic evaluation function.

Surprisingly, LA also appears to *increase* the average number of simulations per tick in some games. An explanation for this can be that, without LA, certain parts of the search tree are estimated to have a low value too quickly. This results in MCTS focusing almost all search effort on a smaller part of the tree, which does not yet have a pessimistic evaluation. MCTS has been implemented to only have a depth limit for the *play-out* step, and not for the *selection* step. Therefore, the result of executing the majority of the simulations in only a small part of the search tree is that simulations in this part of the search tree take a longer amount of time, because they continue for a larger number of ticks. In these games, the addition of LA makes MCTS spread its simulations out more evenly over the search tree, meaning that every individual simulation is shorter on average.

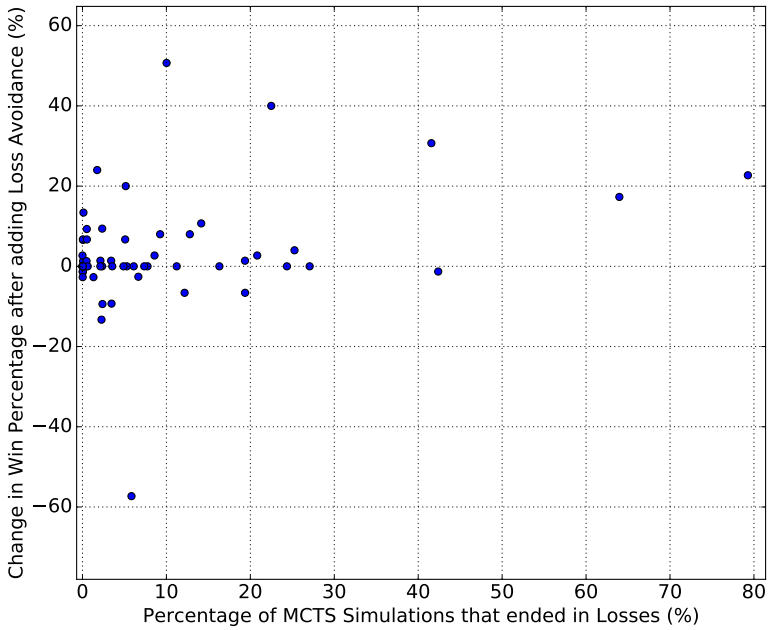


Figure 5.2: Scatter plot with one dot for every game played with Loss Avoidance (LA). The x-axis represents the percentage out of all MCTS simulations in a game that ended in a loss. The y-axis represents the change in win percentage per game after adding LA to the BFTI agent.

Additionally, it has been investigated whether LA is particularly beneficial, in terms of win percentage, in games where many simulations end in losses, and possibly detrimental in other cases. To investigate this, a scatter plot has been created with one dot for each of the 60 games played (depicted in Figure 5.2). The x-coordinate of a dot is the percentage out of all MCTS simulations in the corresponding game that ended in a loss. These percentages were measured with LA enabled. Any simulation that ended in a loss in the play-out step, and therefore started execution of the LA process, was counted as a loss (even if the LA process ended up finding a non-losing state). The y-coordinate of a dot is the change in win percentage for the corresponding game after adding LA to the BFTI agent.

The sample Pearson correlation coefficient of this data is $r = 0.31$, with a p-value of 0.016. This means that it can be said with 95% confidence that there is some positive correlation between the percentage of simulations ending in losses, and the change in win percentage caused by LA. However, this correlation is relatively low. There are also games with a low percentage of simulations ending in losses where LA is still beneficial. An example of such a game is *Overload*, where LA increases the win percentage by 24.0%, but only 1.8% of the MCTS simulations end in losses on average. This is likely because there are only small areas in the map where there are harmful objects and LA can have an effect, and large areas of the map do not have any harmful objects. For future work, it could be worth computing the percentage of simulations ending in losses only over a history of a few recent ticks, instead of over the entire duration of a game. It may be beneficial to disable LA whenever such a percentage is below some threshold. If the percentage of simulations ending in losses is computed over the entire duration of a game, it does not appear to be beneficial to disable LA whenever this percentage is below some threshold. Figure 5.2 indicates that this would also result in disabling LA in many games where it is beneficial.

Table 5.4: Adding Loss Avoidance to the BFTI agent (75 runs per game / 15 runs per level)

	Win Percentage (%)		Avg. Simulations per Tick	
Games	BFTI	LA	BFTI	LA
aliens	100.0 ± 0.0	100.0 ± 0.0	45.5 ± 1.4	52.7 ± 1.9
bait	5.3 ± 5.1	8.0 ± 6.1	20.0 ± 3.2	18.1 ± 5.6
blacksmoke	0.0 ± 0.0	0.0 ± 0.0	9.5 ± 0.5	7.0 ± 0.7
boloadventures	0.0 ± 0.0	0.0 ± 0.0	2.6 ± 0.9	5.1 ± 1.3
boulderchase	13.3 ± 7.7	6.7 ± 5.6	12.2 ± 0.8	13.5 ± 0.8
boulderdash	9.3 ± 6.6	6.7 ± 5.6	8.7 ± 1.1	11.5 ± 1.0
brainman	2.7 ± 3.6	4.0 ± 4.4	22.4 ± 4.1	28.0 ± 4.0
butterflies	98.7 ± 2.6	97.3 ± 3.6	60.2 ± 1.1	68.6 ± 1.7
cakybaky	0.0 ± 0.0	0.0 ± 0.0	19.1 ± 1.9	15.8 ± 2.4
camelRace	17.3 ± 8.6	8.0 ± 6.1	43.7 ± 2.7	45.9 ± 2.8
catapults	4.0 ± 4.4	21.3 ± 9.3	45.4 ± 4.6	19.3 ± 4.0
chase	5.3 ± 5.1	12.0 ± 7.4	30.7 ± 1.1	31.0 ± 1.6
chipschallenge	17.3 ± 8.6	21.3 ± 9.3	19.3 ± 2.5	13.8 ± 4.0
chopper	36.0 ± 10.9	56.0 ± 11.2	5.2 ± 1.5	8.1 ± 1.7
cookmepasta	0.0 ± 0.0	0.0 ± 0.0	26.4 ± 3.8	30.1 ± 4.6
crossfire	1.3 ± 2.6	12.0 ± 7.4	26.0 ± 0.7	24.4 ± 1.0
defender	70.7 ± 10.3	77.3 ± 9.5	30.1 ± 2.0	35.6 ± 2.5
digdug	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.6	6.6 ± 1.4
eggomania	6.7 ± 5.6	14.7 ± 8.0	57.9 ± 1.3	68.2 ± 1.9
enemycitadel	1.3 ± 2.6	2.7 ± 3.6	4.6 ± 1.1	8.0 ± 1.4
escape	0.0 ± 0.0	40.0 ± 11.1	34.5 ± 1.3	25.5 ± 2.2
factorymanager	88.0 ± 7.4	86.7 ± 7.7	12.1 ± 1.4	15.7 ± 1.2
firecaster	0.0 ± 0.0	0.0 ± 0.0	26.9 ± 3.0	32.5 ± 3.7
firestorms	21.3 ± 9.3	14.7 ± 8.0	25.7 ± 0.5	23.5 ± 2.2
frogs	44.0 ± 11.2	66.7 ± 10.7	17.5 ± 2.7	3.9 ± 0.9
gymkhana	1.3 ± 2.6	0.0 ± 0.0	26.9 ± 1.2	16.8 ± 1.4
hungrybirds	34.7 ± 10.8	44.0 ± 11.2	35.0 ± 3.5	39.5 ± 2.8
iceandfire	0.0 ± 0.0	2.7 ± 3.6	53.5 ± 0.6	42.9 ± 2.0
infection	97.3 ± 3.6	100.0 ± 0.0	32.8 ± 0.7	40.8 ± 1.1
intersection	100.0 ± 0.0	100.0 ± 0.0	36.9 ± 1.3	44.4 ± 1.5
jaws	77.3 ± 9.5	20.0 ± 9.1	50.1 ± 5.7	47.7 ± 5.3
labyrinth	9.3 ± 6.6	10.7 ± 7.0	54.0 ± 0.9	55.2 ± 1.1
lasers	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 0.9	4.7 ± 1.0
lasers2	0.0 ± 0.0	0.0 ± 0.0	2.5 ± 1.2	2.4 ± 1.0
lemmings	0.0 ± 0.0	0.0 ± 0.0	53.6 ± 0.9	60.9 ± 0.9
missilecommand	69.3 ± 10.4	78.7 ± 9.3	139.1 ± 3.8	158.8 ± 6.3
modality	25.3 ± 9.8	25.3 ± 9.8	37.9 ± 17.5	37.4 ± 17.7
overload	17.3 ± 8.6	41.3 ± 11.1	13.5 ± 3.9	22.0 ± 4.8
pacman	0.0 ± 0.0	0.0 ± 0.0	8.5 ± 0.5	11.1 ± 0.6
painter	100.0 ± 0.0	100.0 ± 0.0	51.0 ± 8.1	80.0 ± 8.7
plants	4.0 ± 4.4	5.3 ± 5.1	29.5 ± 0.9	32.5 ± 1.0
plaqueattack	94.7 ± 5.1	92.0 ± 6.1	26.3 ± 1.3	29.2 ± 1.7
portals	13.3 ± 7.7	44.0 ± 11.2	32.5 ± 0.8	16.2 ± 1.8
racebet2	68.0 ± 10.6	54.7 ± 11.3	20.7 ± 0.6	22.7 ± 0.6
realportals	0.0 ± 0.0	0.0 ± 0.0	7.1 ± 2.5	2.5 ± 1.5
realsokoban	0.0 ± 0.0	0.0 ± 0.0	23.6 ± 2.2	32.5 ± 3.0
roguelike	0.0 ± 0.0	0.0 ± 0.0	22.5 ± 1.0	26.1 ± 1.0
seaquest	70.7 ± 10.3	61.3 ± 11.0	28.0 ± 2.3	32.9 ± 2.7
sheriff	100.0 ± 0.0	97.3 ± 3.6	31.9 ± 1.3	38.5 ± 1.6
sokoban	20.0 ± 9.1	26.7 ± 10.0	29.5 ± 2.8	31.2 ± 3.3
solarfox	1.3 ± 2.6	2.7 ± 3.6	74.2 ± 3.7	79.3 ± 4.2
superman	0.0 ± 0.0	0.0 ± 0.0	41.4 ± 1.1	45.1 ± 1.5
surround	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
survivezombies	41.3 ± 11.1	49.3 ± 11.3	26.4 ± 0.9	21.3 ± 1.9
tercio	0.0 ± 0.0	0.0 ± 0.0	4.3 ± 2.0	5.3 ± 2.5
thecitadel	5.3 ± 5.1	18.7 ± 8.8	7.0 ± 1.8	14.7 ± 1.9
waitforbreakfast	20.0 ± 9.1	70.7 ± 10.3	35.8 ± 2.8	58.2 ± 5.8
whackamole	100.0 ± 0.0	100.0 ± 0.0	98.0 ± 2.7	103.9 ± 3.9
zelda	32.0 ± 10.6	38.7 ± 11.0	25.7 ± 1.9	30.3 ± 2.4
zenpuzzle	56.0 ± 11.2	56.0 ± 11.2	83.8 ± 8.7	120.6 ± 11.6
Total	30.0 ± 1.3	33.3 ± 1.4	31.0 ± 6.4	33.7 ± 7.6

5.2.4 Novelty-Based Pruning

Table 5.5 shows the results of adding *Novelty-Based Pruning* (NBP) to the BFTI agent, using a threshold of 1 for novelty tests. It increases the overall win percentage by a small, but statistically significant amount from 30.0% to 33.3%. Some of the games where it is particularly beneficial are *Bait*, *HungryBirds*, *Labyrinth*, *Sokoban* and *Zelda*. Except for *Zelda*, these are all "puzzle" games without any NPCs or nondeterministic effects. These games (including *Zelda*) also tend to have many walls that block movement. NBP can frequently prune actions where the avatar moves into such walls and essentially does nothing. As shown by Table 5.2, the IW(1) and YBCriber agents also perform noticeably better than the MCTS-based benchmark agents in these games (except for IW(1) in *Zelda*).

NBP also significantly *increases* the average number of simulations per tick in some games (most noticeably *Bait*, with an increase from 20.0 to 120.3 simulations per tick on average). This is likely because NBP enables MCTS to find a winning sequence of actions more quickly (sometimes even in the 1 second available for initialization at the start of a game). Once such a sequence has been found, the selection step of MCTS will frequently choose to repeat such a sequence. Such a sequence is also expected to be shorter than a (semi-)random sequence which contains "useless" movements into obstacles, and therefore require less processing time.

The most noticeable game where NBP is detrimental is *ChipsChallenge*, where it reduces the win percentage from 17.3% to 1.3%. The first level of this game is depicted in Figure 5.3. The avatar needs to collect the coins to gain increases in score, and ultimately win the game. To pass the large green squares ("doors"), it is necessary to first pick up the smaller green squares ("keys") near the bottom of the map. However, after picking up the keys, the avatar needs to re-visit some previously visited cells to move back to the doors. Such movements back to the same cells are pruned by NBP, which is why NBP is detrimental in this game.

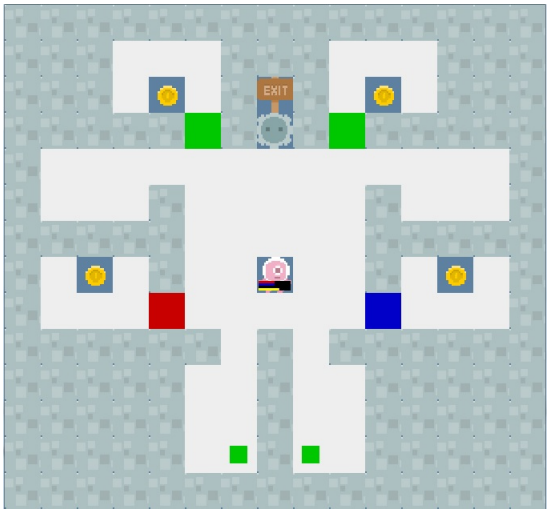


Figure 5.3: The first level of *ChipsChallenge*. The avatar (in the middle) needs to pick up the small green squares near the bottom of the map before he can unlock the larger green squares near the top of the map, and collect the coins behind these green "doors".

Table 5.5: Adding Novelty-Based Pruning to the BFTI agent (75 runs per game / 15 runs per level)

	Win Percentage (%)		Avg. Simulations per Tick	
Games	BFTI	NBP	BFTI	NBP
aliens	100.0 ± 0.0	100.0 ± 0.0	45.5 ± 1.4	42.7 ± 2.0
bait	5.3 ± 5.1	37.3 ± 10.9	20.0 ± 3.2	120.3 ± 27.3
blacksmoke	0.0 ± 0.0	1.3 ± 2.6	9.5 ± 0.5	10.2 ± 0.5
boloadventures	0.0 ± 0.0	0.0 ± 0.0	2.6 ± 0.9	4.7 ± 1.3
boulderchase	13.3 ± 7.7	16.0 ± 8.3	12.2 ± 0.8	14.5 ± 0.8
boulderdash	9.3 ± 6.6	12.0 ± 7.4	8.7 ± 1.1	8.8 ± 1.3
brainman	2.7 ± 3.6	0.0 ± 0.0	22.4 ± 4.1	61.3 ± 3.4
butterflies	98.7 ± 2.6	98.7 ± 2.6	60.2 ± 1.1	65.8 ± 1.2
cakybaky	0.0 ± 0.0	0.0 ± 0.0	19.1 ± 1.9	22.8 ± 3.0
camelRace	17.3 ± 8.6	22.7 ± 9.5	43.7 ± 2.7	61.8 ± 1.5
catapults	4.0 ± 4.4	4.0 ± 4.4	45.4 ± 4.6	45.9 ± 5.2
chase	5.3 ± 5.1	10.7 ± 7.0	30.7 ± 1.1	31.4 ± 1.9
chipschallenge	17.3 ± 8.6	1.3 ± 2.6	19.3 ± 2.5	39.9 ± 2.6
chopper	36.0 ± 10.9	34.7 ± 10.8	5.2 ± 1.5	5.6 ± 1.5
cookmepasta	0.0 ± 0.0	1.3 ± 2.6	26.4 ± 3.8	51.6 ± 4.2
crossfire	1.3 ± 2.6	2.7 ± 3.6	26.0 ± 0.7	26.4 ± 1.2
defender	70.7 ± 10.3	74.7 ± 9.8	30.1 ± 2.0	32.9 ± 2.5
digdug	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.6	10.1 ± 1.4
eggomania	6.7 ± 5.6	10.7 ± 7.0	57.9 ± 1.3	64.6 ± 1.9
enemycitadel	1.3 ± 2.6	1.3 ± 2.6	4.6 ± 1.1	11.6 ± 1.0
escape	0.0 ± 0.0	0.0 ± 0.0	34.5 ± 1.3	29.9 ± 1.9
factorymanager	88.0 ± 7.4	89.3 ± 7.0	12.1 ± 1.4	11.8 ± 1.5
firecaster	0.0 ± 0.0	0.0 ± 0.0	26.9 ± 3.0	58.5 ± 2.7
firestorms	21.3 ± 9.3	17.3 ± 8.6	25.7 ± 0.5	24.1 ± 0.7
frogs	44.0 ± 11.2	37.3 ± 10.9	17.5 ± 2.7	13.1 ± 2.4
gymkhana	1.3 ± 2.6	1.3 ± 2.6	26.9 ± 1.2	27.5 ± 1.9
hungrybirds	34.7 ± 10.8	60.0 ± 11.1	35.0 ± 3.5	84.8 ± 5.9
iceandfire	0.0 ± 0.0	9.3 ± 6.6	53.5 ± 0.6	42.0 ± 1.7
infection	97.3 ± 3.6	96.0 ± 4.4	32.8 ± 0.7	29.9 ± 0.8
intersection	100.0 ± 0.0	100.0 ± 0.0	36.9 ± 1.3	48.7 ± 2.0
jaws	77.3 ± 9.5	74.7 ± 9.8	50.1 ± 5.7	41.5 ± 5.5
labyrinth	9.3 ± 6.6	22.7 ± 9.5	54.0 ± 0.9	43.8 ± 0.8
lasers	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 0.9	4.4 ± 0.9
lasers2	0.0 ± 0.0	0.0 ± 0.0	2.5 ± 1.2	2.6 ± 1.2
lemmings	0.0 ± 0.0	0.0 ± 0.0	53.6 ± 0.9	55.3 ± 0.7
missilecommand	69.3 ± 10.4	84.0 ± 8.3	139.1 ± 3.8	139.5 ± 3.9
modality	25.3 ± 9.8	22.7 ± 9.5	37.9 ± 17.5	73.2 ± 30.8
overload	17.3 ± 8.6	16.0 ± 8.3	13.5 ± 3.9	29.9 ± 3.5
pacman	0.0 ± 0.0	0.0 ± 0.0	8.5 ± 0.5	10.0 ± 0.6
painter	100.0 ± 0.0	100.0 ± 0.0	51.0 ± 8.1	80.3 ± 13.2
plants	4.0 ± 4.4	8.0 ± 6.1	29.5 ± 0.9	27.8 ± 0.7
plaqueattack	94.7 ± 5.1	94.7 ± 5.1	26.3 ± 1.3	26.1 ± 1.0
portals	13.3 ± 7.7	17.3 ± 8.6	32.5 ± 0.8	30.1 ± 1.3
racebet2	68.0 ± 10.6	85.3 ± 8.0	20.7 ± 0.6	21.3 ± 0.6
realportals	0.0 ± 0.0	0.0 ± 0.0	7.1 ± 2.5	8.5 ± 2.8
realsokoban	0.0 ± 0.0	0.0 ± 0.0	23.6 ± 2.2	108.6 ± 7.5
roguelike	0.0 ± 0.0	0.0 ± 0.0	22.5 ± 1.0	27.5 ± 1.0
seaquest	70.7 ± 10.3	82.7 ± 8.6	28.0 ± 2.3	30.8 ± 2.9
sheriff	100.0 ± 0.0	100.0 ± 0.0	31.9 ± 1.3	32.4 ± 1.6
sokoban	20.0 ± 9.1	40.0 ± 11.1	29.5 ± 2.8	81.1 ± 4.9
solarfox	1.3 ± 2.6	2.7 ± 3.6	74.2 ± 3.7	78.4 ± 3.9
superman	0.0 ± 0.0	0.0 ± 0.0	41.4 ± 1.1	53.7 ± 1.2
surround	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	96.0 ± 12.8
survivezombies	41.3 ± 11.1	40.0 ± 11.1	26.4 ± 0.9	22.5 ± 1.3
tercio	0.0 ± 0.0	0.0 ± 0.0	4.3 ± 2.0	7.8 ± 2.0
thecitadel	5.3 ± 5.1	16.0 ± 8.3	7.0 ± 1.8	37.4 ± 3.3
waitforbreakfast	20.0 ± 9.1	22.7 ± 9.5	35.8 ± 2.8	39.7 ± 3.6
whackamole	100.0 ± 0.0	100.0 ± 0.0	98.0 ± 2.7	110.7 ± 4.1
zelda	32.0 ± 10.6	56.0 ± 11.2	25.7 ± 1.9	45.4 ± 0.9
zenpuzzle	56.0 ± 11.2	57.3 ± 11.2	83.8 ± 8.7	200.5 ± 6.3
Total	30.0 ± 1.3	33.0 ± 1.4	31.0 ± 6.4	44.5 ± 9.6

5.2.5 Progressive History and N-Gram Selection Technique

Table 5.6 shows the results obtained by adding *Progressive History* (PH) and/or *N-Gram Selection Technique* (NST) to the BFTI agent. These two enhancements are discussed in the same experiment because they are similar (both introduce action-based biases to different steps of MCTS), and can (partially) share the action-based statistics that they collect. PH was tested with $W = 1$, and NST with $k = 7$, $\epsilon = 0.5$, $N = 3$. These settings are based on (Schuster, 2015). Each of the two enhancements appears to increase the average win percentage by a small (but statistically insignificant) amount, and the combination of the two results in a slightly larger (statistically significant) increase.

Both enhancements appear to be particularly beneficial in *Eggomania*. This is likely because, in this game, the agent should frequently move in the same direction multiple ticks in a row. The enhancements appear to be particularly detrimental in *Frogs*. This is likely because there are many losing game states in the game tree of *Frogs*, which means that many actions will quickly be associated with negative scores by PH and NST. Other than these two cases, it is difficult to characterize in what kinds of games PH and NST are beneficial or detrimental. In most games they are simply expected to slightly increase the quality and reliability of simulations by making the action selection more informed, and therefore also have a small, positive effect on the playing strength.

5.2.6 Tree Reuse

Figure 5.4 depicts the 95% confidence intervals for the average win percentages obtained by adding *Tree Reuse* (TR) to the BFTI agent, for six different values of the decay factor γ. The area shaded in grey is the confidence interval of the win percentage for BFTI without TR. All variants appear to perform better than BFTI, with $\gamma \in \{0.4, 0.6, 1.0\}$ leading to statistically significant increases in win percentage.

Note that TR with $\gamma = 0.0$ is not equivalent to the BFTI agent. Even though it entirely resets any statistics previously collected in nodes, it still keeps the nodes and therefore the structure of the search tree found in previous ticks.

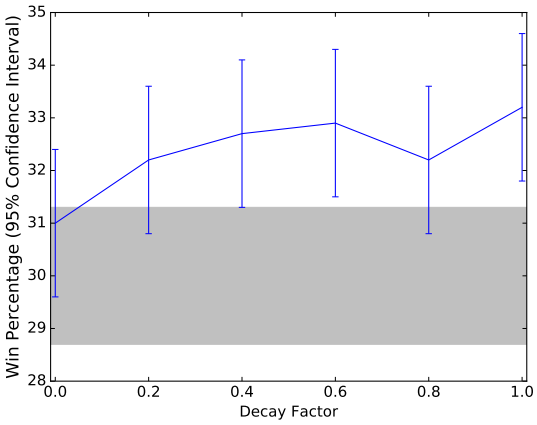


Figure 5.4: 95% confidence intervals for average win percentages of adding Tree Reuse to the BFTI agent with different values for the decay factor γ. The area shaded in grey is the confidence interval for the win percentage of BFTI without TR.

Table 5.6: Adding PH and NST to the BFTI agent (75 runs per game / 15 runs per level)

	Win Percentage (%)				Avg. Simulations per Tick			
Games	BFTI	PH	NST	NST+PH	BFTI	PH	NST	NST+PH
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	45.5 ± 1.4	53.8 ± 1.8	52.0 ± 1.8	51.9 ± 1.7
bait	5.3 ± 5.1	10.7 ± 7.0	14.7 ± 8.0	18.7 ± 8.8	20.0 ± 3.2	26.8 ± 4.6	28.7 ± 4.1	56.2 ± 17.7
blacksmoke	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	9.5 ± 0.5	10.7 ± 0.8	11.3 ± 0.6	11.0 ± 0.6
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.6 ± 0.9	5.1 ± 1.3	4.2 ± 1.0	3.2 ± 1.0
boulderchase	13.3 ± 7.7	16.0 ± 8.3	16.0 ± 8.3	22.7 ± 9.5	12.2 ± 0.8	15.0 ± 1.1	12.0 ± 0.9	13.8 ± 0.7
boulderdash	9.3 ± 6.6	8.0 ± 6.1	12.0 ± 7.4	17.3 ± 8.6	8.7 ± 1.1	9.5 ± 1.3	9.5 ± 1.1	9.2 ± 1.0
brainman	2.7 ± 3.6	5.3 ± 5.1	6.7 ± 5.6	6.7 ± 5.6	22.4 ± 4.1	28.7 ± 4.1	29.8 ± 4.2	22.4 ± 3.6
butterflies	98.7 ± 2.6	100.0 ± 0.0	98.7 ± 2.6	97.3 ± 3.6	60.2 ± 1.1	74.1 ± 1.6	69.2 ± 1.4	67.8 ± 1.3
cakybaky	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	19.1 ± 1.9	23.3 ± 2.2	24.3 ± 1.7	23.1 ± 1.8
camelRace	17.3 ± 8.6	16.0 ± 8.3	17.3 ± 8.6	16.0 ± 8.3	43.7 ± 2.7	48.8 ± 3.0	46.8 ± 3.0	46.7 ± 2.9
catapults	4.0 ± 4.4	5.3 ± 5.1	8.0 ± 6.1	4.0 ± 4.4	45.4 ± 4.6	52.5 ± 5.2	48.4 ± 5.0	47.6 ± 4.9
chase	5.3 ± 5.1	9.3 ± 6.6	6.7 ± 5.6	12.0 ± 7.4	30.7 ± 1.1	37.2 ± 1.5	34.4 ± 1.4	34.0 ± 1.3
chipschallenge	17.3 ± 8.6	17.3 ± 8.6	18.7 ± 8.8	16.0 ± 8.3	19.3 ± 2.5	25.9 ± 2.8	23.6 ± 3.0	24.3 ± 2.3
chopper	36.0 ± 10.9	48.0 ± 11.3	46.7 ± 11.3	49.3 ± 11.3	5.2 ± 1.5	8.3 ± 1.8	7.0 ± 1.7	6.9 ± 1.5
cookmepasta	0.0 ± 0.0	1.3 ± 2.6	1.3 ± 2.6	2.7 ± 3.6	26.4 ± 3.8	29.3 ± 4.3	28.0 ± 3.9	26.1 ± 3.8
crossfire	1.3 ± 2.6	1.3 ± 2.6	2.7 ± 3.6	0.0 ± 0.0	26.0 ± 0.7	31.4 ± 0.7	29.1 ± 0.8	29.9 ± 0.7
defender	70.7 ± 10.3	74.7 ± 9.8	82.7 ± 8.6	66.7 ± 10.7	30.1 ± 2.0	36.6 ± 2.5	30.4 ± 2.0	29.2 ± 1.8
digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.6	6.0 ± 1.7	6.6 ± 1.4	6.2 ± 1.4
eggomania	6.7 ± 5.6	16.0 ± 8.3	16.0 ± 8.3	28.0 ± 10.2	57.9 ± 1.3	69.5 ± 2.1	69.3 ± 1.9	67.3 ± 1.6
enemycitadel	1.3 ± 2.6	1.3 ± 2.6	1.3 ± 2.6	2.7 ± 3.6	4.6 ± 1.1	6.7 ± 1.3	7.1 ± 1.4	6.8 ± 1.3
escape	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 3.6	0.0 ± 0.0	34.5 ± 1.3	36.8 ± 1.5	36.4 ± 1.5	36.9 ± 1.5
factorymanager	88.0 ± 7.4	92.0 ± 6.1	86.7 ± 7.7	90.7 ± 6.6	12.1 ± 1.4	15.6 ± 1.4	12.8 ± 1.2	12.3 ± 1.2
firecaster	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	26.9 ± 3.0	32.4 ± 3.3	29.5 ± 3.1	29.5 ± 3.0
firestorms	21.3 ± 9.3	12.0 ± 7.4	16.0 ± 8.3	17.3 ± 8.6	25.7 ± 0.5	30.7 ± 0.6	29.2 ± 0.6	29.0 ± 0.6
frogs	44.0 ± 11.2	33.3 ± 10.7	22.7 ± 9.5	21.3 ± 9.3	17.5 ± 2.7	21.9 ± 3.0	21.2 ± 1.9	20.5 ± 1.9
gymkhana	1.3 ± 2.6	1.3 ± 2.6	2.7 ± 3.6	5.3 ± 5.1	26.9 ± 1.2	32.4 ± 1.3	31.3 ± 1.0	30.4 ± 1.1
hungrybirds	34.7 ± 10.8	36.0 ± 10.9	32.0 ± 10.6	37.3 ± 10.9	35.0 ± 3.5	37.4 ± 3.4	33.8 ± 3.4	35.4 ± 3.7
iceandfire	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	53.5 ± 0.6	61.6 ± 0.8	55.8 ± 0.9	58.5 ± 0.9
infection	97.3 ± 3.6	100.0 ± 0.0	97.3 ± 3.6	96.0 ± 4.4	32.8 ± 0.7	38.4 ± 0.9	35.4 ± 0.9	34.7 ± 0.8
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	36.9 ± 1.3	44.1 ± 1.2	38.0 ± 1.4	38.7 ± 1.4
jaws	77.3 ± 9.5	76.0 ± 9.7	64.0 ± 10.9	80.0 ± 9.1	50.1 ± 5.7	54.5 ± 7.1	44.3 ± 4.9	45.1 ± 5.2
labyrinth	9.3 ± 6.6	5.3 ± 5.1	5.3 ± 5.1	10.7 ± 7.0	54.0 ± 0.9	59.5 ± 1.1	57.3 ± 1.1	58.5 ± 0.9
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 0.9	3.1 ± 0.8	5.5 ± 0.9	2.9 ± 0.9
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.5 ± 1.2	2.8 ± 1.2	2.6 ± 1.1	2.5 ± 1.1
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	53.6 ± 0.9	61.7 ± 1.0	57.4 ± 0.9	57.8 ± 0.9
missilecommand	69.3 ± 10.4	78.7 ± 9.3	82.7 ± 8.6	84.0 ± 8.3	139.1 ± 3.8	162.8 ± 5.5	138.6 ± 4.1	137.4 ± 4.4
modality	25.3 ± 9.8	26.7 ± 10.0	25.3 ± 9.8	21.3 ± 9.3	37.9 ± 17.5	42.6 ± 21.2	39.4 ± 16.8	31.4 ± 15.6
overload	17.3 ± 8.6	14.7 ± 8.0	41.3 ± 11.1	26.7 ± 10.0	13.5 ± 3.9	17.3 ± 3.7	21.7 ± 3.3	21.4 ± 3.2
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	8.5 ± 0.5	11.9 ± 0.4	10.2 ± 0.5	9.8 ± 0.5
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	51.0 ± 8.1	72.8 ± 8.1	61.8 ± 11.4	53.1 ± 10.1
plants	4.0 ± 4.4	8.0 ± 6.1	6.7 ± 5.6	9.3 ± 6.6	29.5 ± 0.9	32.9 ± 1.2	30.2 ± 1.1	30.4 ± 1.1
plaqueattack	94.7 ± 5.1	84.0 ± 8.3	94.7 ± 5.1	96.0 ± 4.4	26.3 ± 1.3	28.0 ± 1.6	25.8 ± 1.4	24.7 ± 1.5
portals	13.3 ± 7.7	2.7 ± 3.6	14.7 ± 8.0	4.0 ± 4.4	32.5 ± 0.8	39.3 ± 0.8	37.6 ± 0.9	37.0 ± 0.8
racebet2	68.0 ± 10.6	68.0 ± 10.6	88.0 ± 7.4	94.7 ± 5.1	20.7 ± 0.6	24.6 ± 0.7	23.4 ± 0.7	24.5 ± 0.8
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.1 ± 2.5	9.6 ± 2.9	8.0 ± 2.7	7.6 ± 2.7
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	23.6 ± 2.2	30.4 ± 2.3	28.3 ± 2.7	27.2 ± 2.3
roguelike	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	22.5 ± 1.0	26.8 ± 1.0	26.2 ± 1.1	26.9 ± 0.9
seaquest	70.7 ± 10.3	86.7 ± 7.7	78.7 ± 9.3	96.0 ± 4.4	28.0 ± 2.3	30.7 ± 2.3	28.9 ± 3.0	25.9 ± 1.9
sheriff	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	31.9 ± 1.3	38.7 ± 1.8	33.4 ± 1.3	33.5 ± 1.3
sokoban	20.0 ± 9.1	25.3 ± 9.8	30.7 ± 10.4	33.3 ± 10.7	29.5 ± 2.8	36.9 ± 3.7	34.7 ± 3.3	34.4 ± 3.2
solarfox	1.3 ± 2.6	8.0 ± 6.1	4.0 ± 4.4	13.3 ± 7.7	74.2 ± 3.7	92.4 ± 4.5	82.9 ± 4.1	83.3 ± 4.4
superman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	41.4 ± 1.1	48.9 ± 1.6	44.5 ± 1.4	45.7 ± 1.4
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
survivezombies	41.3 ± 11.1	44.0 ± 11.2	46.7 ± 11.3	40.0 ± 11.1	26.4 ± 0.9	30.9 ± 1.1	28.0 ± 1.1	28.4 ± 1.1
tercio	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	4.3 ± 2.0	5.0 ± 2.4	4.9 ± 2.3	4.7 ± 2.4
thecitadel	5.3 ± 5.1	16.0 ± 8.3	14.7 ± 8.0	20.0 ± 9.1	7.0 ± 1.8	15.1 ± 1.7	14.6 ± 1.6	14.4 ± 1.5
waitforbreakfast	20.0 ± 9.1	5.3 ± 5.1	62.7 ± 10.9	48.0 ± 11.3	35.8 ± 2.8	40.8 ± 3.1	61.4 ± 5.1	46.6 ± 3.7
whackamole	100.0 ± 0.0	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	98.0 ± 2.7	122.9 ± 3.0	109.4 ± 2.7	112.2 ± 2.7
zelda	32.0 ± 10.6	30.7 ± 10.4	36.0 ± 10.9	25.3 ± 9.8	25.7 ± 1.9	28.4 ± 2.0	29.0 ± 1.9	26.3 ± 1.9
zenpuzzle	56.0 ± 11.2	57.3 ± 11.2	50.7 ± 11.3	58.7 ± 11.1	83.8 ± 8.7	108.2 ± 9.8	46.1 ± 8.4	38.1 ± 1.7
Total	30.0 ± 1.3	30.7 ± 1.3	32.6 ± 1.4	33.2 ± 1.4	31.0 ± 6.4	37.2 ± 7.7	33.9 ± 6.5	33.4 ± 6.5

5.2.7 Knowledge-Based Evaluations

Table 5.7 shows the results of adding *Knowledge-Based Evaluations* (KBE) to the BFTI agent. It significantly increases the average win percentage, from 30.0% to 36.1%. It appears to be beneficial in a wide variety of games. This is because almost all games in the GVG-AI framework are implemented in a way where, ultimately, games are won or lost upon collisions between objects. This is exactly what the heuristic evaluation function of KBE is based on.

The most notable game where KBE is detrimental is *ChipsChallenge*, which was previously discussed in the experiment with NBP. NBP was described to be detrimental in this game because it prevents the agent from picking up a key and moving towards a corresponding door within the same simulation. KBE is detrimental for a different reason, which is that there is no direct score gain associated with picking up the keys. Therefore, the weight learned for key objects by KBE converges to 0, meaning that the agent will not attempt to move towards the key. Because the (locked) doors are also not recognized as obstacles, KBE instead rewards the agent for moving as close as possible to the coins, but does not enable MCTS to find a way through the door.

5.2.8 Temporal-Difference Tree Search

Table 5.8 shows the win percentages obtained by adding a number of different variants of *Temporal-Difference Tree Search* to the BFTI agent. The following variants have been tested in this experiment:

- **TDIR(λ)**: The Sarsa-UCT(λ) algorithm implemented exactly as given by the pseudocode in Subsection 4.2.6, taking **i**ntermediate **r**ewards into account. Tested for $\lambda = 0.5$ and $\lambda = 0.8$.
- **TDMS(λ)**: A variant that does not take intermediate rewards into account. In addition to any changes described in Subsection 4.2.6, this variant also uses the V values backed up through TD backups for the final **m**ove **s**election at the end of every tick. This is different from the other variants, which only use the V values from TD backups in the selection step, but still use the regular X values for the final move selection. Tested for $\lambda = 0.5$ and $\lambda = 0.8$.
- **TD(λ)**: Does not take intermediate rewards into account and does not use the V values of TD backups for the final move selection. This variant is the one that is most similar to the implementation described by Vodopivec (2016). However, there are still a number of differences, because this variant is implemented on top of the BFTI agent, whereas Vodopivec (2016) implemented Sarsa-UCT(λ) in the Sample Open-Loop MCTS controller. Tested for $\lambda = 0.5$ and $\lambda = 0.8$ (Vodopivec (2016) used $\lambda = 0.8$).

Most of these variants appear to provide small, but statistically insignificant increases in overall performance. In these variants, $\lambda = 0.5$ appears to perform better than $\lambda = 0.8$. The most basic TD(λ) variant appears to outperform the others.

Finally, Table 5.9 shows the results of a **TDKBE(λ)** variant, which uses *Knowledge-Based Evaluations* (KBE) for intermediate rewards. The difference between this variant and TDIR is that TDIR frequently has intermediate rewards of 0, because it only takes real changes in game score into account as intermediate rewards. This variant is compared to the KBE agent, instead of the BFTI agent, because it also incorporates the changes of the KBE enhancement. This variant is detrimental with respect to the KBE agent, and this can easily be explained by the columns showing the average number of simulations per tick. The pathfinding necessary for using KBE is simply too expensive computationally to use for every state generated in a simulation.

Table 5.7: Adding Knowledge-Based Evaluations to the BFTI agent (75 runs per game / 15 runs per level)

	Win Percentage (%)		Avg. Simulations per Tick	
Games	BFTI	KBE	BFTI	KBE
aliens	100.0 ± 0.0	100.0 ± 0.0	45.5 ± 1.4	53.4 ± 1.6
bait	5.3 ± 5.1	0.0 ± 0.0	20.0 ± 3.2	19.1 ± 3.0
blacksmoke	0.0 ± 0.0	2.7 ± 3.6	9.5 ± 0.5	8.0 ± 0.7
boloadventures	0.0 ± 0.0	0.0 ± 0.0	2.6 ± 0.9	2.2 ± 0.7
boulderchase	13.3 ± 7.7	22.7 ± 9.5	12.2 ± 0.8	12.2 ± 1.2
boulderdash	9.3 ± 6.6	17.3 ± 8.6	8.7 ± 1.1	9.1 ± 0.8
brainman	2.7 ± 3.6	0.0 ± 0.0	22.4 ± 4.1	17.7 ± 3.2
butterflies	98.7 ± 2.6	98.7 ± 2.6	60.2 ± 1.1	72.0 ± 2.4
cakybaky	0.0 ± 0.0	1.3 ± 2.6	19.1 ± 1.9	25.8 ± 2.2
camelRace	17.3 ± 8.6	96.0 ± 4.4	43.7 ± 2.7	50.9 ± 2.5
catapults	4.0 ± 4.4	5.3 ± 5.1	45.4 ± 4.6	52.9 ± 5.3
chase	5.3 ± 5.1	14.7 ± 8.0	30.7 ± 1.1	35.1 ± 1.7
chipschallenge	17.3 ± 8.6	1.3 ± 2.6	19.3 ± 2.5	17.2 ± 2.3
chopper	36.0 ± 10.9	58.7 ± 11.1	5.2 ± 1.5	8.5 ± 1.7
cookmepasta	0.0 ± 0.0	0.0 ± 0.0	26.4 ± 3.8	24.2 ± 4.2
crossfire	1.3 ± 2.6	13.3 ± 7.7	26.0 ± 0.7	29.9 ± 1.1
defender	70.7 ± 10.3	73.3 ± 10.0	30.1 ± 2.0	34.8 ± 2.5
digdug	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.6	5.7 ± 1.6
eggomania	6.7 ± 5.6	62.7 ± 10.9	57.9 ± 1.3	57.9 ± 1.7
enemycitadel	1.3 ± 2.6	1.3 ± 2.6	4.6 ± 1.1	6.1 ± 1.3
escape	0.0 ± 0.0	0.0 ± 0.0	34.5 ± 1.3	37.2 ± 1.4
factorymanager	88.0 ± 7.4	100.0 ± 0.0	12.1 ± 1.4	15.7 ± 1.2
firecaster	0.0 ± 0.0	0.0 ± 0.0	26.9 ± 3.0	26.8 ± 2.6
firestorms	21.3 ± 9.3	33.3 ± 10.7	25.7 ± 0.5	28.0 ± 0.5
frogs	44.0 ± 11.2	36.0 ± 10.9	17.5 ± 2.7	22.7 ± 2.5
gymkhana	1.3 ± 2.6	0.0 ± 0.0	26.9 ± 1.2	30.3 ± 1.5
hungrybirds	34.7 ± 10.8	64.0 ± 10.9	35.0 ± 3.5	32.3 ± 4.9
iceandfire	0.0 ± 0.0	0.0 ± 0.0	53.5 ± 0.6	54.7 ± 0.9
infection	97.3 ± 3.6	100.0 ± 0.0	32.8 ± 0.7	37.8 ± 1.1
intersection	100.0 ± 0.0	100.0 ± 0.0	36.9 ± 1.3	38.3 ± 1.3
jaws	77.3 ± 9.5	66.7 ± 10.7	50.1 ± 5.7	54.0 ± 7.0
labyrinth	9.3 ± 6.6	18.7 ± 8.8	54.0 ± 0.9	59.7 ± 2.2
lasers	0.0 ± 0.0	0.0 ± 0.0	3.1 ± 0.9	5.2 ± 1.4
lasers2	0.0 ± 0.0	0.0 ± 0.0	2.5 ± 1.2	4.5 ± 1.6
lemmings	0.0 ± 0.0	0.0 ± 0.0	53.6 ± 0.9	58.0 ± 0.9
missilecommand	69.3 ± 10.4	100.0 ± 0.0	139.1 ± 3.8	145.6 ± 7.9
modality	25.3 ± 9.8	26.7 ± 10.0	37.9 ± 17.5	41.8 ± 20.2
overload	17.3 ± 8.6	50.7 ± 11.3	13.5 ± 3.9	25.9 ± 3.1
pacman	0.0 ± 0.0	0.0 ± 0.0	8.5 ± 0.5	11.7 ± 0.5
painter	100.0 ± 0.0	100.0 ± 0.0	51.0 ± 8.1	70.6 ± 9.9
plants	4.0 ± 4.4	1.3 ± 2.6	29.5 ± 0.9	36.7 ± 1.1
plaqueattack	94.7 ± 5.1	86.7 ± 7.7	26.3 ± 1.3	27.9 ± 1.5
portals	13.3 ± 7.7	12.0 ± 7.4	32.5 ± 0.8	38.5 ± 1.2
racebet2	68.0 ± 10.6	65.3 ± 10.8	20.7 ± 0.6	20.6 ± 0.8
realportals	0.0 ± 0.0	0.0 ± 0.0	7.1 ± 2.5	19.4 ± 2.3
realsokoban	0.0 ± 0.0	0.0 ± 0.0	23.6 ± 2.2	25.1 ± 3.3
roguelike	0.0 ± 0.0	0.0 ± 0.0	22.5 ± 1.0	22.6 ± 1.3
seaquest	70.7 ± 10.3	80.0 ± 9.1	28.0 ± 2.3	33.0 ± 2.6
sheriff	100.0 ± 0.0	100.0 ± 0.0	31.9 ± 1.3	39.5 ± 1.5
sokoban	20.0 ± 9.1	17.3 ± 8.6	29.5 ± 2.8	30.1 ± 3.3
solarfox	1.3 ± 2.6	12.0 ± 7.4	74.2 ± 3.7	87.5 ± 4.3
superman	0.0 ± 0.0	1.3 ± 2.6	41.4 ± 1.1	43.5 ± 1.5
surround	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
survivezombies	41.3 ± 11.1	41.3 ± 11.1	26.4 ± 0.9	29.1 ± 1.1
tercio	0.0 ± 0.0	0.0 ± 0.0	4.3 ± 2.0	3.7 ± 2.1
thecitadel	5.3 ± 5.1	10.7 ± 7.0	7.0 ± 1.8	10.0 ± 1.9
waitforbreakfast	20.0 ± 9.1	52.0 ± 11.3	35.8 ± 2.8	54.4 ± 4.6
whackamole	100.0 ± 0.0	100.0 ± 0.0	98.0 ± 2.7	116.0 ± 3.0
zelda	32.0 ± 10.6	66.7 ± 10.7	25.7 ± 1.9	24.5 ± 2.0
zenpuzzle	56.0 ± 11.2	52.0 ± 11.3	83.8 ± 8.7	58.7 ± 4.9
Total	30.0 ± 1.3	36.1 ± 1.4	31.0 ± 6.4	34.4 ± 6.9

Table 5.8: Adding Temporal-Difference Tree Search to the BFTI agent (75 runs per game / 15 runs per level)

	Win Percentage (%)						
Games	BFTI	TDIR(0.5)	TDIR(0.8)	TDMS(0.5)	TDMS(0.8)	TD(0.5)	TD(0.8)
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
bait	5.3 ± 5.1	5.3 ± 5.1	8.0 ± 6.1	5.3 ± 5.1	8.0 ± 6.1	4.0 ± 4.4	12.0 ± 7.4
blacksmoke	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 3.6	2.7 ± 3.6	1.3 ± 2.6	0.0 ± 0.0
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
boulderchase	13.3 ± 7.7	16.0 ± 8.3	10.7 ± 7.0	16.0 ± 8.3	14.7 ± 8.0	16.0 ± 8.3	18.7 ± 8.8
boulderdash	9.3 ± 6.6	8.0 ± 6.1	6.7 ± 5.6	6.7 ± 5.6	8.0 ± 6.1	10.7 ± 7.0	8.0 ± 6.1
brainman	2.7 ± 3.6	5.3 ± 5.1	6.7 ± 5.6	6.7 ± 5.6	6.7 ± 5.6	6.7 ± 5.6	4.0 ± 4.4
butterflies	98.7 ± 2.6	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	100.0 ± 0.0	93.3 ± 5.6	98.7 ± 2.6
cakybaky	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
camelRace	17.3 ± 8.6	9.3 ± 6.6	14.7 ± 8.0	17.3 ± 8.6	16.0 ± 8.3	8.0 ± 6.1	13.3 ± 7.7
catapults	4.0 ± 4.4	9.3 ± 6.6	6.7 ± 5.6	2.7 ± 3.6	0.0 ± 0.0	14.7 ± 8.0	5.3 ± 5.1
chase	5.3 ± 5.1	8.0 ± 6.1	8.0 ± 6.1	6.7 ± 5.6	5.3 ± 5.1	8.0 ± 6.1	5.3 ± 5.1
chipschallenge	17.3 ± 8.6	18.7 ± 8.8	17.3 ± 8.6	16.0 ± 8.3	20.0 ± 9.1	18.7 ± 8.8	20.0 ± 9.1
chopper	36.0 ± 10.9	33.3 ± 10.7	29.3 ± 10.3	38.7 ± 11.0	40.0 ± 11.1	40.0 ± 11.1	36.0 ± 10.9
cookmepasta	0.0 ± 0.0	1.3 ± 2.6	4.0 ± 4.4	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
crossfire	1.3 ± 2.6	4.0 ± 4.4	4.0 ± 4.4	2.7 ± 3.6	2.7 ± 3.6	5.3 ± 5.1	2.7 ± 3.6
defender	70.7 ± 10.3	76.0 ± 9.7	64.0 ± 10.9	76.0 ± 9.7	74.7 ± 9.8	74.7 ± 9.8	80.0 ± 9.1
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	0.0 ± 0.0
eggomania	6.7 ± 5.6	8.0 ± 6.1	9.3 ± 6.6	17.3 ± 8.6	16.0 ± 8.3	12.0 ± 7.4	14.7 ± 8.0
enemycitadel	1.3 ± 2.6	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 3.6	0.0 ± 0.0
escape	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
factorymanager	88.0 ± 7.4	86.7 ± 7.7	89.3 ± 7.0	85.3 ± 8.0	93.3 ± 5.6	92.0 ± 6.1	96.0 ± 4.4
firecaster	0.0 ± 0.0	2.7 ± 3.6	2.7 ± 3.6	2.7 ± 3.6	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
firestorms	21.3 ± 9.3	13.3 ± 7.7	18.7 ± 8.8	24.0 ± 9.7	12.0 ± 7.4	10.7 ± 7.0	12.0 ± 7.4
frogs	44.0 ± 11.2	34.7 ± 10.8	41.3 ± 11.1	21.3 ± 9.3	18.7 ± 8.8	40.0 ± 11.1	49.3 ± 11.3
gymkhana	1.3 ± 2.6	2.7 ± 3.6	0.0 ± 0.0	1.3 ± 2.6	2.7 ± 3.6	5.3 ± 5.1	0.0 ± 0.0
hungrybirds	34.7 ± 10.8	34.7 ± 10.8	34.7 ± 10.8	30.7 ± 10.4	34.7 ± 10.8	30.7 ± 10.4	40.0 ± 11.1
iceandfire	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
infection	97.3 ± 3.6	96.0 ± 4.4	94.7 ± 5.1	98.7 ± 2.6	100.0 ± 0.0	98.7 ± 2.6	98.7 ± 2.6
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
jaws	77.3 ± 9.5	72.0 ± 10.2	64.0 ± 10.9	58.7 ± 11.1	66.7 ± 10.7	62.7 ± 10.9	65.3 ± 10.8
labyrinth	9.3 ± 6.6	9.3 ± 6.6	6.7 ± 5.6	1.3 ± 2.6	9.3 ± 6.6	8.0 ± 6.1	8.0 ± 6.1
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
missilecommand	69.3 ± 10.4	72.0 ± 10.2	70.7 ± 10.3	72.0 ± 10.2	74.7 ± 9.8	78.7 ± 9.3	76.0 ± 9.7
modality	25.3 ± 9.8	24.0 ± 9.7	26.7 ± 10.0	26.7 ± 10.0	29.3 ± 10.3	28.0 ± 10.2	26.7 ± 10.0
overload	17.3 ± 8.6	18.7 ± 8.8	14.7 ± 8.0	21.3 ± 9.3	21.3 ± 9.3	22.7 ± 9.5	25.3 ± 9.8
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
plants	4.0 ± 4.4	8.0 ± 6.1	4.0 ± 4.4	9.3 ± 6.6	5.3 ± 5.1	6.7 ± 5.6	5.3 ± 5.1
plaqueattack	94.7 ± 5.1	92.0 ± 6.1	94.7 ± 5.1	97.3 ± 3.6	89.3 ± 7.0	92.0 ± 6.1	96.0 ± 4.4
portals	13.3 ± 7.7	16.0 ± 8.3	14.7 ± 8.0	17.3 ± 8.6	18.7 ± 8.8	18.7 ± 8.8	13.3 ± 7.7
racebet2	68.0 ± 10.6	61.3 ± 11.0	58.7 ± 11.1	60.0 ± 11.1	65.3 ± 10.8	62.7 ± 10.9	65.3 ± 10.8
realportals	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
roguelike	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
seaquest	70.7 ± 10.3	77.3 ± 9.5	81.3 ± 8.8	72.0 ± 10.2	81.3 ± 8.8	82.7 ± 8.6	77.3 ± 9.5
sheriff	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
sokoban	20.0 ± 9.1	24.0 ± 9.7	24.0 ± 9.7	22.7 ± 9.5	18.7 ± 8.8	26.7 ± 10.0	20.0 ± 9.1
solarfox	1.3 ± 2.6	1.3 ± 2.6	1.3 ± 2.6	0.0 ± 0.0	4.0 ± 4.4	4.0 ± 4.4	4.0 ± 4.4
superman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
survivezombies	41.3 ± 11.1	46.7 ± 11.3	44.0 ± 11.2	42.7 ± 11.2	44.0 ± 11.2	40.0 ± 11.1	44.0 ± 11.2
tercio	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
thecitadel	5.3 ± 5.1	12.0 ± 7.4	5.3 ± 5.1	10.7 ± 7.0	4.0 ± 4.4	18.7 ± 8.8	13.3 ± 7.7
waitforbreakfast	20.0 ± 9.1	36.0 ± 10.9	29.3 ± 10.3	65.3 ± 10.8	12.0 ± 7.4	29.3 ± 10.3	14.7 ± 8.0
whackamole	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	98.7 ± 2.6
zelda	32.0 ± 10.6	38.7 ± 11.0	36.0 ± 10.9	34.7 ± 10.8	30.7 ± 10.4	33.3 ± 10.7	22.7 ± 9.5
zenpuzzle	56.0 ± 11.2	56.0 ± 11.2	56.0 ± 11.2	60.0 ± 11.1	60.0 ± 11.1	64.0 ± 10.9	64.0 ± 10.9
Total	30.0 ± 1.3	30.7 ± 1.3	30.0 ± 1.3	30.8 ± 1.3	30.2 ± 1.3	31.2 ± 1.4	30.9 ± 1.4

Table 5.9: Knowledge-Based Intermediate Evaluations in TDTS (75 runs per game / 15 runs per level)

	Win Percentage (%)			Avg. Simulations per Tick		
Games	KBE	TDKBE(0.5)	TDKBE(0.8)	KBE	TDKBE(0.5)	TDKBE(0.8)
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	53.4 ± 1.6	41.7 ± 1.2	41.2 ± 1.3
bait	0.0 ± 0.0	1.3 ± 2.6	2.7 ± 3.6	19.1 ± 3.0	16.4 ± 2.9	17.2 ± 2.5
blacksmoke	2.7 ± 3.6	0.0 ± 0.0	0.0 ± 0.0	8.0 ± 0.7	4.3 ± 0.4	4.2 ± 0.5
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.2 ± 0.7	0.8 ± 0.3	1.3 ± 0.5
boulderchase	22.7 ± 9.5	34.7 ± 10.8	33.3 ± 10.7	12.2 ± 1.2	10.1 ± 0.7	9.4 ± 0.9
boulderdash	17.3 ± 8.6	2.7 ± 3.6	8.0 ± 6.1	9.1 ± 0.8	5.8 ± 0.7	4.7 ± 0.7
brainman	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 3.6	17.7 ± 3.2	6.1 ± 2.2	7.2 ± 2.2
butterflies	98.7 ± 2.6	93.3 ± 5.6	80.0 ± 9.1	72.0 ± 2.4	34.9 ± 1.4	26.5 ± 1.1
cakybaky	1.3 ± 2.6	2.7 ± 3.6	0.0 ± 0.0	25.8 ± 2.2	20.9 ± 2.1	21.2 ± 2.0
camelRace	96.0 ± 4.4	25.3 ± 9.8	13.3 ± 7.7	50.9 ± 2.5	31.9 ± 3.1	30.0 ± 3.0
catapults	5.3 ± 5.1	18.7 ± 8.8	13.3 ± 7.7	52.9 ± 5.3	32.6 ± 3.4	33.8 ± 3.0
chase	14.7 ± 8.0	4.0 ± 4.4	5.3 ± 5.1	35.1 ± 1.7	27.9 ± 0.9	25.8 ± 0.8
chipschallenge	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	17.2 ± 2.3	14.8 ± 1.6	14.0 ± 1.7
chopper	58.7 ± 11.1	25.3 ± 9.8	28.0 ± 10.2	8.5 ± 1.7	3.8 ± 1.0	3.7 ± 1.0
cookmepasta	0.0 ± 0.0	2.7 ± 3.6	0.0 ± 0.0	24.2 ± 4.2	20.0 ± 3.1	23.5 ± 3.4
crossfire	13.3 ± 7.7	34.7 ± 10.8	29.3 ± 10.3	29.9 ± 1.1	19.7 ± 0.8	19.5 ± 0.9
defender	73.3 ± 10.0	69.3 ± 10.4	69.3 ± 10.4	34.8 ± 2.5	24.8 ± 1.6	25.1 ± 1.7
digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	5.7 ± 1.6	3.9 ± 1.2	3.6 ± 1.2
eggomania	62.7 ± 10.9	56.0 ± 11.2	76.0 ± 9.7	57.9 ± 1.7	46.9 ± 1.8	46.1 ± 1.8
enemycitadel	1.3 ± 2.6	2.7 ± 3.6	1.3 ± 2.6	6.1 ± 1.3	3.6 ± 1.1	3.9 ± 1.2
escape	0.0 ± 0.0	1.3 ± 2.6	2.7 ± 3.6	37.2 ± 1.4	29.5 ± 1.0	30.6 ± 1.0
factorymanager	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	15.7 ± 1.2	9.2 ± 1.1	9.2 ± 1.1
firecaster	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 4.4	26.8 ± 2.6	20.6 ± 2.1	20.8 ± 2.2
firestorms	33.3 ± 10.7	18.7 ± 8.8	28.0 ± 10.2	28.0 ± 0.5	18.5 ± 0.5	19.1 ± 0.7
frogs	36.0 ± 10.9	46.7 ± 11.3	37.3 ± 10.9	22.7 ± 2.5	15.9 ± 1.9	13.2 ± 2.1
gymkhana	0.0 ± 0.0	1.3 ± 2.6	2.7 ± 3.6	30.3 ± 1.5	12.4 ± 0.5	12.9 ± 0.4
hungrybirds	64.0 ± 10.9	10.7 ± 7.0	5.3 ± 5.1	32.3 ± 4.9	32.7 ± 2.6	31.9 ± 2.1
iceandfire	0.0 ± 0.0	0.0 ± 0.0	1.3 ± 2.6	54.7 ± 0.9	28.8 ± 1.1	29.8 ± 1.2
infection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	37.8 ± 1.1	25.7 ± 1.2	24.9 ± 1.2
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	38.3 ± 1.3	29.4 ± 0.8	30.5 ± 0.8
jaws	66.7 ± 10.7	46.7 ± 11.3	56.0 ± 11.2	54.0 ± 7.0	43.5 ± 4.7	45.5 ± 4.5
labyrinth	18.7 ± 8.8	16.0 ± 8.3	10.7 ± 7.0	59.7 ± 2.2	44.2 ± 1.3	44.0 ± 1.2
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	5.2 ± 1.4	3.1 ± 0.9	3.3 ± 0.8
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	4.5 ± 1.6	2.7 ± 1.0	2.5 ± 1.0
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	58.0 ± 0.9	38.0 ± 1.2	37.3 ± 1.2
missilecommand	100.0 ± 0.0	88.0 ± 7.4	81.3 ± 8.8	145.6 ± 7.9	105.3 ± 3.9	104.3 ± 4.3
modality	26.7 ± 10.0	25.3 ± 9.8	24.0 ± 9.7	41.8 ± 20.2	27.1 ± 12.0	29.3 ± 11.8
overload	50.7 ± 11.3	54.7 ± 11.3	54.7 ± 11.3	25.9 ± 3.1	15.1 ± 1.8	14.1 ± 2.2
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	11.7 ± 0.5	7.1 ± 0.3	6.8 ± 0.3
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	70.6 ± 9.9	54.8 ± 6.0	65.0 ± 7.6
plants	1.3 ± 2.6	4.0 ± 4.4	6.7 ± 5.6	36.7 ± 1.1	26.5 ± 0.8	25.2 ± 0.8
plaqueattack	86.7 ± 7.7	93.3 ± 5.6	92.0 ± 6.1	27.9 ± 1.5	21.2 ± 1.1	21.1 ± 1.0
portals	12.0 ± 7.4	18.7 ± 8.8	22.7 ± 9.5	38.5 ± 1.2	28.9 ± 0.9	27.6 ± 1.2
racebet2	65.3 ± 10.8	62.7 ± 10.9	52.0 ± 11.3	20.6 ± 0.8	16.2 ± 0.6	17.8 ± 0.6
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	19.4 ± 2.3	13.7 ± 2.4	12.4 ± 2.6
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	25.1 ± 3.3	33.2 ± 2.5	29.7 ± 2.1
roguelike	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	22.6 ± 1.3	13.4 ± 1.0	12.4 ± 1.0
seaquest	80.0 ± 9.1	81.3 ± 8.8	73.3 ± 10.0	33.0 ± 2.6	27.4 ± 2.1	28.5 ± 2.1
sheriff	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	39.5 ± 1.5	27.0 ± 0.9	26.9 ± 1.0
sokoban	17.3 ± 8.6	22.7 ± 9.5	20.0 ± 9.1	30.1 ± 3.3	29.0 ± 2.1	32.8 ± 1.5
solarfox	12.0 ± 7.4	4.0 ± 4.4	4.0 ± 4.4	87.5 ± 4.3	69.5 ± 3.5	69.3 ± 3.7
superman	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	43.5 ± 1.5	37.2 ± 0.8	36.2 ± 0.9
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
survivezombies	41.3 ± 11.1	38.7 ± 11.0	41.3 ± 11.1	29.1 ± 1.1	19.9 ± 0.7	18.7 ± 0.8
tercio	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	3.7 ± 2.1	3.1 ± 1.8	3.4 ± 1.9
thecitadel	10.7 ± 7.0	8.0 ± 6.1	6.7 ± 5.6	10.0 ± 1.9	8.5 ± 2.0	8.7 ± 2.0
waitforbreakfast	52.0 ± 11.3	32.0 ± 10.6	54.7 ± 11.3	54.4 ± 4.6	34.8 ± 2.4	37.6 ± 2.3
whackamole	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	116.0 ± 3.0	92.2 ± 1.8	95.6 ± 2.1
zelda	66.7 ± 10.7	93.3 ± 5.6	96.0 ± 4.4	24.5 ± 2.0	29.4 ± 1.3	30.1 ± 1.2
zenpuzzle	52.0 ± 11.3	61.3 ± 11.0	58.7 ± 11.1	58.7 ± 4.9	60.8 ± 1.3	56.1 ± 1.8
Total	36.1 ± 1.4	33.4 ± 1.4	33.3 ± 1.4	34.4 ± 6.9	25.4 ± 5.3	25.5 ± 5.4

5.2.9 Enhancements Combined

Combining all Enhancements

Table 5.10 shows the results of an experiment intended to evaluate the performance that can be obtained by combining all enhancements discussed in this thesis. The following three agents are included in this table:

- **Enh. (No TDTS)**: An agent with all enhancements described in this thesis except for TDTS. This variant was included because TDTS was not found to provide a statistically significant increase in win percentage, and not assumed to have any other important side-effects (like BFTI).
- **Enh. TD(0.5)**: An agent with all enhancements described in this thesis, including backups from the Sarsa-UCT(λ) algorithm with $\lambda = 0.5$. Because of the results described in Subsection 5.2.8, this agent was implemented not to take intermediate rewards into account, and not to use the values from TD backups for the final move selection.
- **Enh. TD(0.8)**: A similar agent as described above, but using $\lambda = 0.8$.

These agents include the *Deterministic Game Detection* (DGD) enhancement, which has not yet been evaluated individually. This is because some parts of DGD are only relevant if TR or NBP are included as well. The impact of DGD is evaluated at the end of this subsection by disabling it in the MCTS agent with other enhancements enabled.

This table shows that the combination of enhancements described in this thesis leads to a significant increase in the average win percentage over sixty different games, from 30.0% to 47.2%−48.4% in comparison to the BFTI agent. This is not yet sufficient to beat the 52.4% of YBCriber (one of the winning agents of 2015), but it is relatively close. The inclusion of TDTS appears to be slightly detrimental (both for $\lambda = 0.5$ and $\lambda = 0.8$), but the difference in win percentage is not statistically significant.

Variations in Implementation Details of KBE and TR+BFTI

Table 5.11 shows the results of an experiment where small implementation details are changed in the KBE enhancement, and in the interaction between TR and BFTI. These are tested in the **Enh. TD(λ)** agents described above, for both $\lambda = 0.5$ and $\lambda = 0.8$. Both of these agents are also included again for comparison. The following two variations are tested:

- **Always KBE**: Subsection 4.2.4 describes that a final state s_T of a simulation is only evaluated using the knowledge-based heuristic evaluation if the regular evaluation $X(s_T)$ is equal to the evaluation of the root state s_0; $X(s_0) = X(s_T)$. In this variation, the knowledge-based heuristic evaluation is used even if $X(s_0) \neq X(s_T)$.
- **No TR-BFTI**: Subsection 4.2.3 describes that, when the new root node at the start of a new tick is already fully expanded due to TR, the BFTI process is still executed because *safety prepruning* may potentially help to reduce the branching factor. In this variation, this is not done. This means that the BFTI process is only executed if the new root at the start of a new tick is not yet fully expanded.

This table shows that the "Always KBE" variant performs at least similarly to the standard implementation, and possibly better. Not executing the BFTI process when a new root node is already fully expanded due to TR appears to be detrimental. Neither of these variations affect the win percentage by a statistically significant amount.

Table 5.10: All Enhancements combined (with and without TDTS) (75 runs per game / 15 runs per level)

	Win Percentage (%)			Avg. Simulations per Tick		
Games	Enh. (No TDTS)	Enh. TD(0.5)	Enh. TD(0.8)	Enh. (No TDTS)	Enh. TD(0.5)	Enh. TD(0.8)
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	33.5 ± 1.4	33.3 ± 1.3	33.3 ± 1.3
bait	37.3 ± 10.9	32.0 ± 10.6	34.7 ± 10.8	75.4 ± 28.1	65.2 ± 24.6	69.7 ± 26.7
blacksmoke	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	7.2 ± 0.4	4.0 ± 0.4	4.3 ± 0.4
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.8 ± 0.4	2.3 ± 0.5	1.4 ± 0.3
boulderchase	41.3 ± 11.1	40.0 ± 11.1	42.7 ± 11.2	13.0 ± 0.3	10.8 ± 0.5	11.0 ± 0.5
boulderdash	20.0 ± 9.1	14.7 ± 8.0	13.3 ± 7.7	7.8 ± 0.6	6.9 ± 0.6	7.8 ± 0.7
brainman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	34.9 ± 3.4	34.7 ± 3.1	34.1 ± 3.1
butterflies	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	53.3 ± 1.0	52.9 ± 1.1	51.9 ± 0.9
cakybaky	10.7 ± 7.0	16.0 ± 8.3	10.7 ± 7.0	19.8 ± 1.9	15.6 ± 2.1	15.1 ± 2.1
camelRace	100.0 ± 0.0	98.7 ± 2.6	93.3 ± 5.6	46.2 ± 0.9	45.3 ± 1.0	46.4 ± 0.8
catapults	28.0 ± 10.2	26.7 ± 10.0	24.0 ± 9.7	19.6 ± 2.9	17.2 ± 2.2	15.4 ± 2.1
chase	41.3 ± 11.1	32.0 ± 10.6	29.3 ± 10.3	31.3 ± 1.4	32.7 ± 1.4	30.2 ± 1.5
chipschallenge	0.0 ± 0.0	4.0 ± 4.4	1.3 ± 2.6	16.2 ± 2.1	12.5 ± 2.2	12.4 ± 2.3
chopper	92.0 ± 6.1	90.7 ± 6.6	89.3 ± 7.0	7.1 ± 1.0	5.5 ± 0.8	5.5 ± 0.7
cookmepasta	16.0 ± 8.3	16.0 ± 8.3	14.7 ± 8.0	28.4 ± 3.5	25.9 ± 3.0	25.6 ± 3.4
crossfire	78.7 ± 9.3	85.3 ± 8.0	89.3 ± 7.0	25.5 ± 0.9	21.7 ± 0.6	22.5 ± 0.7
defender	70.7 ± 10.3	74.7 ± 9.8	62.7 ± 10.9	21.1 ± 1.3	17.9 ± 1.5	18.1 ± 1.7
digdug	1.3 ± 2.6	0.0 ± 0.0	1.3 ± 2.6	7.5 ± 0.5	8.5 ± 0.5	7.7 ± 0.5
eggomania	69.3 ± 10.4	54.7 ± 11.3	57.3 ± 11.2	39.0 ± 0.8	34.2 ± 0.9	34.3 ± 1.0
enemycitadel	2.7 ± 3.6	1.3 ± 2.6	4.0 ± 4.4	9.9 ± 0.6	5.1 ± 1.1	4.4 ± 1.0
escape	92.0 ± 6.1	81.3 ± 8.8	85.3 ± 8.0	51.1 ± 5.6	19.9 ± 1.6	21.9 ± 3.3
factorymanager	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	10.5 ± 1.1	9.8 ± 0.9	10.1 ± 0.9
firecaster	0.0 ± 0.0	5.3 ± 5.1	2.7 ± 3.6	18.6 ± 1.9	19.1 ± 1.7	17.7 ± 1.5
firestorms	78.7 ± 9.3	61.3 ± 11.0	65.3 ± 10.8	14.4 ± 1.1	10.8 ± 1.0	10.7 ± 1.0
frogs	48.0 ± 11.3	48.0 ± 11.3	45.3 ± 11.3	6.0 ± 1.1	6.5 ± 0.9	6.9 ± 1.0
gymkhana	10.7 ± 7.0	6.7 ± 5.6	2.7 ± 3.6	15.7 ± 0.9	8.8 ± 0.4	8.9 ± 0.4
hungrybirds	97.3 ± 3.6	94.7 ± 5.1	97.3 ± 3.6	70.4 ± 4.9	39.9 ± 4.0	61.9 ± 2.5
iceandfire	77.3 ± 9.5	80.0 ± 9.1	72.0 ± 10.2	32.7 ± 1.7	27.5 ± 1.6	26.8 ± 1.7
infection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	24.8 ± 0.6	21.8 ± 0.4	22.1 ± 0.4
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	28.7 ± 0.9	25.8 ± 1.4	25.1 ± 1.4
jaws	18.7 ± 8.8	16.0 ± 8.3	20.0 ± 9.1	30.4 ± 3.3	31.0 ± 3.9	28.5 ± 3.5
labyrinth	100.0 ± 0.0	98.7 ± 2.6	98.7 ± 2.6	69.2 ± 3.1	48.9 ± 3.0	49.6 ± 2.9
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.2 ± 0.4	3.6 ± 0.6	2.2 ± 0.4
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 0.8	1.9 ± 0.6	2.0 ± 0.6
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	39.4 ± 0.5	36.1 ± 0.5	36.6 ± 0.5
missilecommand	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	102.5 ± 3.7	89.3 ± 3.6	87.7 ± 4.2
modality	49.3 ± 11.3	52.0 ± 11.3	53.3 ± 11.3	50.5 ± 22.0	58.2 ± 22.4	54.4 ± 22.8
overload	88.0 ± 7.4	89.3 ± 7.0	77.3 ± 9.5	21.9 ± 1.5	17.4 ± 1.6	17.2 ± 1.3
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 0.4	3.6 ± 0.4	4.6 ± 0.4
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	45.4 ± 9.3	50.5 ± 7.0	63.8 ± 6.2
plants	1.3 ± 2.6	6.7 ± 5.6	5.3 ± 5.1	23.3 ± 0.5	21.7 ± 0.5	21.6 ± 0.5
plaqueattack	100.0 ± 0.0	92.0 ± 6.1	90.7 ± 6.6	20.5 ± 0.7	18.4 ± 0.8	18.0 ± 0.8
portals	77.3 ± 9.5	77.3 ± 9.5	70.7 ± 10.3	25.3 ± 1.3	18.8 ± 1.4	20.0 ± 1.8
racebet2	86.7 ± 7.7	93.3 ± 5.6	93.3 ± 5.6	16.9 ± 0.4	15.8 ± 0.3	15.7 ± 0.3
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	6.8 ± 1.6	6.7 ± 1.5	5.1 ± 1.4
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	17.1 ± 2.0	26.2 ± 4.5	32.7 ± 5.4
roguelike	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	18.4 ± 0.5	17.9 ± 0.4	17.1 ± 0.7
seaquest	56.0 ± 11.2	50.7 ± 11.3	64.0 ± 10.9	24.1 ± 2.4	22.7 ± 3.2	18.4 ± 2.2
sheriff	98.7 ± 2.6	97.3 ± 3.6	97.3 ± 3.6	23.4 ± 0.9	21.5 ± 0.9	21.4 ± 0.8
sokoban	44.0 ± 11.2	40.0 ± 11.1	37.3 ± 10.9	24.4 ± 1.9	17.9 ± 1.3	18.4 ± 1.3
solarfox	6.7 ± 5.6	1.3 ± 2.6	8.0 ± 6.1	58.1 ± 3.3	59.1 ± 3.5	53.9 ± 3.0
superman	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	31.0 ± 1.0	29.4 ± 1.0	31.1 ± 0.9
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	145.0 ± 5.5	141.8 ± 5.7
survivezombies	45.3 ± 11.3	34.7 ± 10.8	44.0 ± 11.2	15.4 ± 1.2	11.8 ± 1.2	12.4 ± 1.3
tercio	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.4	5.7 ± 1.3	5.1 ± 1.3
thecitadel	36.0 ± 10.9	25.3 ± 9.8	30.7 ± 10.4	21.0 ± 1.7	12.5 ± 1.7	14.6 ± 2.1
waitforbreakfast	60.0 ± 11.1	64.0 ± 10.9	58.7 ± 11.1	32.7 ± 3.6	43.7 ± 4.5	29.4 ± 2.2
whackamole	100.0 ± 0.0	97.3 ± 3.6	100.0 ± 0.0	61.2 ± 2.5	54.6 ± 2.1	55.2 ± 2.3
zelda	52.0 ± 11.3	90.7 ± 6.6	70.7 ± 10.3	32.0 ± 0.6	36.5 ± 0.6	35.9 ± 0.6
zenpuzzle	64.0 ± 10.9	61.3 ± 11.0	70.7 ± 10.3	43.8 ± 7.6	31.2 ± 2.6	28.2 ± 2.6
Total	48.4 ± 1.5	47.5 ± 1.5	47.2 ± 1.5	27.4 ± 5.3	26.7 ± 6.2	26.8 ± 6.2

Table 5.11: Small variations in the implementation of KBE and the combination of TR + BFTI

	Win Percentage (%)					
Games	Enh. TD(0.5)	Always KBE (0.5)	No TR-BFTI (0.5)	Enh. TD(0.8)	Always KBE (0.8)	No TR-BFTI (0.8)
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
bait	32.0 ± 10.6	34.7 ± 10.8	37.3 ± 10.9	34.7 ± 10.8	36.0 ± 10.9	30.7 ± 10.4
blacksmoke	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
boulderchase	40.0 ± 11.1	40.0 ± 11.1	34.7 ± 10.8	42.7 ± 11.2	41.3 ± 11.1	41.3 ± 11.1
boulderdash	14.7 ± 8.0	18.7 ± 8.8	16.0 ± 8.3	13.3 ± 7.7	16.0 ± 8.3	21.3 ± 9.3
brainman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
butterflies	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
cakybaky	16.0 ± 8.3	9.3 ± 6.6	12.0 ± 7.4	10.7 ± 7.0	13.3 ± 7.7	13.3 ± 7.7
camelRace	98.7 ± 2.6	96.0 ± 4.4	98.7 ± 2.6	93.3 ± 5.6	94.7 ± 5.1	98.7 ± 2.6
catapults	26.7 ± 10.0	24.0 ± 9.7	25.3 ± 9.8	24.0 ± 9.7	25.3 ± 9.8	24.0 ± 9.7
chase	32.0 ± 10.6	28.0 ± 10.2	34.7 ± 10.8	29.3 ± 10.3	30.7 ± 10.4	16.0 ± 8.3
chipschallenge	4.0 ± 4.4	5.3 ± 5.1	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0
chopper	90.7 ± 6.6	92.0 ± 6.1	89.3 ± 7.0	89.3 ± 7.0	85.3 ± 8.0	86.5 ± 7.8
cookmepasta	16.0 ± 8.3	8.0 ± 6.1	9.3 ± 6.6	14.7 ± 8.0	13.3 ± 7.7	10.7 ± 7.0
crossfire	85.3 ± 8.0	85.3 ± 8.0	78.7 ± 9.3	89.3 ± 7.0	85.3 ± 8.0	77.3 ± 9.5
defender	74.7 ± 9.8	73.3 ± 10.0	73.3 ± 10.0	62.7 ± 10.9	76.0 ± 9.7	74.7 ± 9.8
digdug	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.3 ± 2.6	1.3 ± 2.6	0.0 ± 0.0
eggomania	54.7 ± 11.3	60.0 ± 11.1	60.0 ± 11.1	57.3 ± 11.2	77.3 ± 9.5	60.0 ± 11.1
enemycitadel	1.3 ± 2.6	1.3 ± 2.6	0.0 ± 0.0	4.0 ± 4.4	2.7 ± 3.6	4.0 ± 4.4
escape	81.3 ± 8.8	89.3 ± 7.0	88.0 ± 7.4	85.3 ± 8.0	82.7 ± 8.6	88.0 ± 7.4
factorymanager	98.7 ± 2.6	100.0 ± 0.0	98.7 ± 2.6	100.0 ± 0.0	97.3 ± 3.6	100.0 ± 0.0
firecaster	5.3 ± 5.1	2.7 ± 3.6	9.3 ± 6.6	2.7 ± 3.6	5.3 ± 5.1	4.0 ± 4.4
firestorms	61.3 ± 11.0	69.3 ± 10.4	58.7 ± 11.1	65.3 ± 10.8	70.7 ± 10.3	68.0 ± 10.6
frogs	48.0 ± 11.3	50.7 ± 11.3	42.7 ± 11.2	45.3 ± 11.3	54.7 ± 11.3	50.7 ± 11.3
gymkhana	6.7 ± 5.6	2.7 ± 3.6	6.7 ± 5.6	2.7 ± 3.6	6.7 ± 5.6	9.3 ± 6.6
hungrybirds	94.7 ± 5.1	97.3 ± 3.6	97.3 ± 3.6	97.3 ± 3.6	97.3 ± 3.6	97.3 ± 3.6
iceandfire	80.0 ± 9.1	78.7 ± 9.3	78.7 ± 9.3	72.0 ± 10.2	76.0 ± 9.7	77.3 ± 9.5
infection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
jaws	16.0 ± 8.3	24.0 ± 9.7	18.7 ± 8.8	20.0 ± 9.1	17.3 ± 8.6	17.3 ± 8.6
labyrinth	98.7 ± 2.6	98.7 ± 2.6	97.3 ± 3.6	98.7 ± 2.6	100.0 ± 0.0	96.0 ± 4.4
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
missilecommand	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
modality	52.0 ± 11.3	58.7 ± 11.1	53.3 ± 11.3	53.3 ± 11.3	58.7 ± 11.1	49.3 ± 11.3
overload	89.3 ± 7.0	78.7 ± 9.3	89.3 ± 7.0	77.3 ± 9.5	81.3 ± 8.8	84.0 ± 8.3
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
plants	6.7 ± 5.6	2.7 ± 3.6	1.3 ± 2.6	5.3 ± 5.1	5.3 ± 5.1	0.0 ± 0.0
plaqueattack	92.0 ± 6.1	94.7 ± 5.1	94.7 ± 5.1	90.7 ± 6.6	96.0 ± 4.4	94.7 ± 5.1
portals	77.3 ± 9.5	74.7 ± 9.8	74.7 ± 9.8	70.7 ± 10.3	74.7 ± 9.8	74.7 ± 9.8
racebet2	93.3 ± 5.6	92.0 ± 6.1	97.3 ± 3.6	93.3 ± 5.6	89.3 ± 7.0	94.7 ± 5.1
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
realsokoban	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
roguelike	0.0 ± 0.0	4.0 ± 4.4	2.7 ± 3.6	0.0 ± 0.0	2.7 ± 3.6	0.0 ± 0.0
seaquest	50.7 ± 11.3	73.3 ± 10.0	60.0 ± 11.1	64.0 ± 10.9	65.3 ± 10.8	56.0 ± 11.2
sheriff	97.3 ± 3.6	97.3 ± 3.6	100.0 ± 0.0	97.3 ± 3.6	97.3 ± 3.6	97.3 ± 3.6
sokoban	40.0 ± 11.1	41.3 ± 11.1	37.3 ± 10.9	37.3 ± 10.9	32.0 ± 10.6	37.3 ± 10.9
solarfox	1.3 ± 2.6	4.0 ± 4.4	2.7 ± 3.6	8.0 ± 6.1	6.7 ± 5.6	9.3 ± 6.6
superman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
survivezombies	34.7 ± 10.8	40.0 ± 11.1	41.3 ± 11.1	44.0 ± 11.2	41.3 ± 11.1	42.7 ± 11.2
tercio	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0
thecitadel	25.3 ± 9.8	22.7 ± 9.5	18.7 ± 8.8	30.7 ± 10.4	24.0 ± 9.7	21.3 ± 9.3
waitforbreakfast	64.0 ± 10.9	62.7 ± 10.9	65.3 ± 10.8	58.7 ± 11.1	61.3 ± 11.0	62.7 ± 10.9
whackamole	97.3 ± 3.6	94.7 ± 5.1	96.0 ± 4.4	100.0 ± 0.0	94.7 ± 5.1	96.0 ± 4.4
zelda	90.7 ± 6.6	58.7 ± 11.1	33.3 ± 10.7	70.7 ± 10.3	96.0 ± 4.4	65.3 ± 10.8
zenpuzzle	61.3 ± 11.0	62.7 ± 10.9	62.7 ± 10.9	70.7 ± 10.3	64.0 ± 10.9	64.0 ± 10.9
Total	47.5 ± 1.5	47.5 ± 1.5	46.6 ± 1.5	47.2 ± 1.5	48.3 ± 1.5	46.9 ± 1.5

Deactivating BFTI or DGD

In the final experiment, BFTI and DGD are each individually disabled in an agent with all other enhancements enabled except for TDTS. This experiment was performed before TDTS was implemented. It was not repeated anymore afterwards, because TDTS appeared to be slightly detrimental when combined with all other enhancements in the previous experiments. The purpose of testing an agent with BFTI disabled is to test the assumption made in Subsection 5.2.2 that BFTI may enable other enhancements to find more wins, even though BFTI itself appeared to be slightly detrimental in that experiment. The purpose of testing an agent with DGD deactivated is to investigate the impact that DGD has on the agent's performance. This was not tested in an agent without any other enhancements because some of the features of DGD are only relevant when other enhancements are enabled.

The results of this experiment are shown in Table 5.12. Deactivating BFTI appears to slightly decrease the average win percentage over all sixty games. This difference is not a statistically significant amount, but still interesting considering the *inclusion* of BFTI appeared to decrease the average win percentage (again by a statistically insignificant amount) when other enhancements were absent. This provides some evidence for the assumption that the positive effect that BFTI has on survival duration allows other enhancements to find more winning lines of play. BFTI appears to be particularly beneficial in *Boulderchase*, *Chase*, *Firestorms*, and *Sheriff*. The first two of these have enemies that actively chase the avatar, and the last two have many projectiles that the avatar should dodge. In these games, the safety prepruning of BFTI can frequently prune some actions.

Deactivating DGD leads to a significant decrease in the average win percentage. This indicates that DGD has a significant positive effect on the win percentage, increasing it from 41.6% to 48.4%. The largest effects are observed in *Catapults*, *CookMePasta*, *Escape*, *HungryBirds*, *IceAndFire*, *Labyrinth* and *Modality*. These have all been manually inspected and confirmed to be deterministic games, which are the types of games where DGD is expected to be beneficial.

In the games of *Boulderchase* and *Chopper*, DGD also appears to improve the win percentage according to these results. These are nondeterministic games, and have also correctly been classified by DGD as nondeterministic. Therefore, DGD would be expected to have no impact in these games at best, or a slightly detrimental effect due to the computational overhead of classifying the games at worst. In *Boulderchase*, the confidence intervals still overlap slightly. In *Chopper*, this is more difficult to explain. The table shows that the average number of simulations per tick was significantly lower for the "**No DGD**" agent in this experiment, which can explain the observed difference in win percentage. However, this difference in the average number of simulations is difficult to explain. On different hardware with a faster processor (2.67 GHz instead of 2.20 GHz), the difference in win percentage was not reproducible, and the agents with or without DGD performed similarly in *Chopper*. A possible side-effect of DGD is that it could reduce the size of the tree created by MCTS in the 1 second of initialization start at the start of every game, due to the computational overhead of game classification. It is possible that this tree created by the "**No DGD**" agent is too large. A large tree causes subsequent MCTS simulations to be slower, because MCTS has been implemented to only use a depth limit for the play-out step (and not for the selection step, which continues for a longer amount of time in a larger tree). On slower hardware, this could cause MCTS to be unable to complete any simulations in the subsequent 40-millisecond ticks, and play the *Nil* action due to running out of processing time. In *Chopper* this can easily cause losses by getting hit by projectiles.

In Table 5.13 and Table 5.14, the results of the "**Enh. (No TDTS)**" and "**No DGD**" agents are compared once more, with the games split up in a set of deterministic games and a set of nondeterministic games, respectively. Table 5.13 shows that DGD significantly increases the win percentage in deterministic games, from 27.5% to 40.5%. Table 5.14 only shows a small, statistically insignificant increase of the average win percentage in nondeterminstic games. This indicates that any side-effects that cause the observed difference in performance in *Chopper* do not significantly affect all nondeterministic games.

Table 5.12: Deactivating BFTI or DGD (75 runs per game / 15 runs per level)

	Win Percentage (%)			Avg. Simulations per Tick		
Games	Enh. (No TDTS)	No BFTI	No DGD	Enh. (No TDTS)	No BFTI	No DGD
aliens	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	33.5 ± 1.4	36.2 ± 1.2	31.0 ± 1.5
bait	37.3 ± 10.9	32.0 ± 10.6	25.3 ± 9.8	75.4 ± 28.1	88.8 ± 32.9	79.5 ± 27.0
blacksmoke	1.3 ± 2.6	1.3 ± 2.6	1.3 ± 2.6	7.2 ± 0.4	8.0 ± 0.4	4.7 ± 0.4
boloadventures	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	1.8 ± 0.4	4.4 ± 0.5	0.4 ± 0.2
boulderchase	41.3 ± 11.1	26.7 ± 10.0	21.3 ± 9.3	13.0 ± 0.3	12.8 ± 0.4	9.4 ± 0.5
boulderdash	20.0 ± 9.1	13.3 ± 7.7	14.7 ± 8.0	7.8 ± 0.6	12.1 ± 0.3	8.1 ± 0.6
brainman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	34.9 ± 3.4	34.6 ± 3.3	38.3 ± 2.3
butterflies	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	53.3 ± 1.0	56.4 ± 1.0	53.8 ± 1.0
cakybaky	10.7 ± 7.0	6.7 ± 5.6	9.3 ± 6.6	19.8 ± 1.9	20.4 ± 2.0	15.1 ± 2.4
camelRace	100.0 ± 0.0	97.3 ± 3.6	96.0 ± 4.4	46.2 ± 0.9	47.9 ± 1.0	43.9 ± 1.3
catapults	28.0 ± 10.2	34.7 ± 10.8	14.7 ± 8.0	19.6 ± 2.9	21.7 ± 3.4	17.7 ± 3.0
chase	41.3 ± 11.1	26.7 ± 10.0	22.7 ± 9.5	31.3 ± 1.4	34.9 ± 1.3	31.3 ± 1.5
chipschallenge	0.0 ± 0.0	1.3 ± 2.6	0.0 ± 0.0	16.2 ± 2.1	17.6 ± 1.9	28.2 ± 1.7
chopper	92.0 ± 6.1	80.0 ± 9.1	30.7 ± 10.4	7.1 ± 1.0	9.9 ± 0.6	2.8 ± 1.0
cookmepasta	16.0 ± 8.3	17.3 ± 8.6	2.7 ± 3.6	28.4 ± 3.5	28.2 ± 3.2	40.3 ± 2.3
crossfire	78.7 ± 9.3	82.7 ± 8.6	86.7 ± 7.7	25.5 ± 0.9	26.1 ± 0.8	23.9 ± 0.7
defender	70.7 ± 10.3	68.0 ± 10.6	74.7 ± 9.8	21.1 ± 1.3	21.9 ± 1.5	19.3 ± 1.6
digdug	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	7.5 ± 0.5	9.2 ± 0.5	2.5 ± 1.0
eggomania	69.3 ± 10.4	60.0 ± 11.1	61.3 ± 11.0	39.0 ± 0.8	42.2 ± 1.1	38.3 ± 0.7
enemycitadel	2.7 ± 3.6	1.3 ± 2.6	2.7 ± 3.6	9.9 ± 0.6	10.0 ± 0.5	5.3 ± 1.2
escape	92.0 ± 6.1	89.3 ± 7.0	12.0 ± 7.4	51.1 ± 5.6	61.7 ± 7.2	29.6 ± 1.5
factorymanager	100.0 ± 0.0	100.0 ± 0.0	98.7 ± 2.6	10.5 ± 1.1	11.2 ± 1.2	4.2 ± 1.1
firecaster	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	18.6 ± 1.9	19.5 ± 2.0	19.0 ± 1.7
firestorms	78.7 ± 9.3	58.7 ± 11.1	69.3 ± 10.4	14.4 ± 1.1	14.2 ± 1.1	11.5 ± 1.1
frogs	48.0 ± 11.3	42.7 ± 11.2	49.3 ± 11.3	6.0 ± 1.1	9.4 ± 1.1	7.3 ± 1.2
gymkhana	10.7 ± 7.0	9.3 ± 6.6	6.7 ± 5.6	15.7 ± 0.9	16.2 ± 0.8	13.7 ± 0.7
hungrybirds	97.3 ± 3.6	97.3 ± 3.6	76.0 ± 9.7	70.4 ± 4.9	76.3 ± 5.6	9.2 ± 4.3
iceandfire	77.3 ± 9.5	78.7 ± 9.3	6.7 ± 5.6	32.7 ± 1.7	34.0 ± 1.7	26.6 ± 1.2
infection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	24.8 ± 0.6	25.2 ± 0.7	23.7 ± 0.5
intersection	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	28.7 ± 0.9	30.0 ± 1.0	26.8 ± 1.2
jaws	18.7 ± 8.8	24.0 ± 9.7	21.3 ± 9.3	30.4 ± 3.3	31.8 ± 3.9	28.9 ± 4.1
labyrinth	100.0 ± 0.0	98.7 ± 2.6	58.7 ± 11.1	69.2 ± 3.1	48.9 ± 3.3	54.9 ± 3.2
lasers	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.2 ± 0.4	5.9 ± 0.6	0.8 ± 0.4
lasers2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 0.8	5.2 ± 0.6	1.3 ± 0.7
lemmings	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	39.4 ± 0.5	39.4 ± 0.5	38.3 ± 0.5
missilecommand	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	102.5 ± 3.7	102.1 ± 3.6	99.1 ± 4.1
modality	49.3 ± 11.3	52.0 ± 11.3	20.0 ± 9.1	50.5 ± 22.0	71.6 ± 26.8	79.3 ± 28.7
overload	88.0 ± 7.4	82.7 ± 8.6	81.3 ± 8.8	21.9 ± 1.5	20.2 ± 1.2	17.5 ± 1.3
pacman	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 0.4	8.5 ± 0.4	3.1 ± 0.5
painter	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	45.4 ± 9.3	60.3 ± 12.3	51.3 ± 16.7
plants	1.3 ± 2.6	1.3 ± 2.6	2.7 ± 3.6	23.3 ± 0.5	23.9 ± 0.5	22.6 ± 0.5
plaqueattack	100.0 ± 0.0	96.0 ± 4.4	93.3 ± 5.6	20.5 ± 0.7	20.5 ± 0.7	18.5 ± 0.8
portals	77.3 ± 9.5	74.7 ± 9.8	74.7 ± 9.8	25.3 ± 1.3	21.8 ± 1.2	23.9 ± 1.2
racebet2	86.7 ± 7.7	88.0 ± 7.4	90.7 ± 6.6	16.9 ± 0.4	17.8 ± 0.4	15.1 ± 0.4
realportals	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	6.8 ± 1.6	7.3 ± 1.6	6.0 ± 1.8
realsokoban	0.0 ± 0.0	2.7 ± 3.6	0.0 ± 0.0	17.1 ± 2.0	17.5 ± 1.5	58.7 ± 5.4
roguelike	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 3.6	18.4 ± 0.5	18.8 ± 0.8	18.4 ± 0.4
seaquest	56.0 ± 11.2	60.0 ± 11.1	57.3 ± 11.2	24.1 ± 2.4	22.6 ± 1.8	19.2 ± 2.3
sheriff	98.7 ± 2.6	88.0 ± 7.4	100.0 ± 0.0	23.4 ± 0.9	24.5 ± 1.0	21.9 ± 0.8
sokoban	44.0 ± 11.2	40.0 ± 11.1	50.7 ± 11.3	24.4 ± 1.9	23.6 ± 2.5	77.5 ± 4.7
solarfox	6.7 ± 5.6	4.0 ± 4.4	6.7 ± 5.6	58.1 ± 3.3	58.9 ± 2.7	55.6 ± 3.3
superman	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	31.0 ± 1.0	30.9 ± 1.2	30.8 ± 1.0
surround	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	132.3 ± 7.6	1.0 ± 0.0
survivezombies	45.3 ± 11.3	36.0 ± 10.9	37.3 ± 10.9	15.4 ± 1.2	14.8 ± 1.3	11.4 ± 1.3
tercio	1.3 ± 2.6	0.0 ± 0.0	0.0 ± 0.0	4.8 ± 1.4	7.0 ± 1.2	4.6 ± 1.4
thecitadel	36.0 ± 10.9	22.7 ± 9.5	21.3 ± 9.3	21.0 ± 1.7	20.7 ± 1.3	31.4 ± 3.5
waitforbreakfast	60.0 ± 11.1	65.3 ± 10.8	61.3 ± 11.0	32.7 ± 3.6	27.8 ± 1.9	27.4 ± 2.1
whackamole	100.0 ± 0.0	98.7 ± 2.6	98.7 ± 2.6	61.2 ± 2.5	62.5 ± 2.5	57.9 ± 2.3
zelda	52.0 ± 11.3	93.3 ± 5.6	77.3 ± 9.5	32.0 ± 0.6	33.0 ± 0.5	31.6 ± 0.5
zenpuzzle	64.0 ± 10.9	74.7 ± 9.8	54.7 ± 11.3	43.8 ± 7.6	173.7 ± 4.3	27.3 ± 6.3
Total	48.4 ± 1.5	47.1 ± 1.5	41.6 ± 1.4	27.4 ± 5.3	33.2 ± 8.0	26.7 ± 5.7

Table 5.13: DGD in Deterministic Games (75 runs per game / 15 runs per level)

	Win Percentage (%)		Avg. Simulations per Tick	
Games	Enh. (No TDTS)	No DGD	Enh. (No TDTS)	No DGD
bait	37.3 ± 10.9	25.3 ± 9.8	75.4 ± 28.1	79.5 ± 27.0
boloadventures	0.0 ± 0.0	0.0 ± 0.0	1.8 ± 0.4	0.4 ± 0.2
brainman	0.0 ± 0.0	0.0 ± 0.0	34.9 ± 3.4	38.3 ± 2.3
catapults	28.0 ± 10.2	14.7 ± 8.0	19.6 ± 2.9	17.7 ± 3.0
chipschallenge	0.0 ± 0.0	0.0 ± 0.0	16.2 ± 2.1	28.2 ± 1.7
cookmepasta	16.0 ± 8.3	2.7 ± 3.6	28.4 ± 3.5	40.3 ± 2.3
digdug	1.3 ± 2.6	0.0 ± 0.0	7.5 ± 0.5	2.5 ± 1.0
escape	92.0 ± 6.1	12.0 ± 7.4	51.1 ± 5.6	29.6 ± 1.5
factorymanager	100.0 ± 0.0	98.7 ± 2.6	10.5 ± 1.1	4.2 ± 1.1
hungrybirds	97.3 ± 3.6	76.0 ± 9.7	70.4 ± 4.9	9.2 ± 4.3
iceandfire	77.3 ± 9.5	6.7 ± 5.6	32.7 ± 1.7	26.6 ± 1.2
labyrinth	100.0 ± 0.0	58.7 ± 11.1	69.2 ± 3.1	54.9 ± 3.2
lasers	0.0 ± 0.0	0.0 ± 0.0	2.2 ± 0.4	0.8 ± 0.4
lasers2	0.0 ± 0.0	0.0 ± 0.0	2.7 ± 0.8	1.3 ± 0.7
modality	49.3 ± 11.3	20.0 ± 9.1	50.5 ± 22.0	79.3 ± 28.7
painter	100.0 ± 0.0	100.0 ± 0.0	45.4 ± 9.3	51.3 ± 16.7
racebet2	86.7 ± 7.7	90.7 ± 6.6	16.9 ± 0.4	15.1 ± 0.4
realportals	0.0 ± 0.0	0.0 ± 0.0	6.8 ± 1.6	6.0 ± 1.8
realsokoban	0.0 ± 0.0	0.0 ± 0.0	17.1 ± 2.0	58.7 ± 5.4
sokoban	44.0 ± 11.2	50.7 ± 11.3	24.4 ± 1.9	77.5 ± 4.7
tercio	1.3 ± 2.6	0.0 ± 0.0	4.8 ± 1.4	4.6 ± 1.4
thecitadel	36.0 ± 10.9	21.3 ± 9.3	21.0 ± 1.7	31.4 ± 3.5
zenpuzzle	64.0 ± 10.9	54.7 ± 11.3	43.8 ± 7.6	27.3 ± 6.3
Total	40.5 ± 2.3	27.5 ± 2.1	28.4 ± 9.6	29.8 ± 11.0

Table 5.14: DGD in Nondeterminstic Games (75 runs per game / 15 runs per level)

	Win Percentage (%)		Avg. Simulations per Tick	
Games	Enh. (No TDTS)	No DGD	Enh. (No TDTS)	No DGD
aliens	100.0 ± 0.0	100.0 ± 0.0	33.5 ± 1.4	31.0 ± 1.5
blacksmoke	1.3 ± 2.6	1.3 ± 2.6	7.2 ± 0.4	4.7 ± 0.4
boulderchase	41.3 ± 11.1	21.3 ± 9.3	13.0 ± 0.3	9.4 ± 0.5
boulderdash	20.0 ± 9.1	14.7 ± 8.0	7.8 ± 0.6	8.1 ± 0.6
butterflies	100.0 ± 0.0	100.0 ± 0.0	53.3 ± 1.0	53.8 ± 1.0
cakybaky	10.7 ± 7.0	9.3 ± 6.6	19.8 ± 1.9	15.1 ± 2.4
camelRace	100.0 ± 0.0	96.0 ± 4.4	46.2 ± 0.9	43.9 ± 1.3
chase	41.3 ± 11.1	22.7 ± 9.5	31.3 ± 1.4	31.3 ± 1.5
chopper	92.0 ± 6.1	30.7 ± 10.4	7.1 ± 1.0	2.8 ± 1.0
crossfire	78.7 ± 9.3	86.7 ± 7.7	25.5 ± 0.9	23.9 ± 0.7
defender	70.7 ± 10.3	74.7 ± 9.8	21.1 ± 1.3	19.3 ± 1.6
eggomania	69.3 ± 10.4	61.3 ± 11.0	39.0 ± 0.8	38.3 ± 0.7
enemycitadel	2.7 ± 3.6	2.7 ± 3.6	9.9 ± 0.6	5.3 ± 1.2
firecaster	0.0 ± 0.0	0.0 ± 0.0	18.6 ± 1.9	19.0 ± 1.7
firestorms	78.7 ± 9.3	69.3 ± 10.4	14.4 ± 1.1	11.5 ± 1.1
frogs	48.0 ± 11.3	49.3 ± 11.3	6.0 ± 1.1	7.3 ± 1.2
gymkhana	10.7 ± 7.0	6.7 ± 5.6	15.7 ± 0.9	13.7 ± 0.7
infection	100.0 ± 0.0	100.0 ± 0.0	24.8 ± 0.6	23.7 ± 0.5
intersection	100.0 ± 0.0	100.0 ± 0.0	28.7 ± 0.9	26.8 ± 1.2
jaws	18.7 ± 8.8	21.3 ± 9.3	30.4 ± 3.3	28.9 ± 4.1
lemmings	0.0 ± 0.0	0.0 ± 0.0	39.4 ± 0.5	38.3 ± 0.5
missilecommand	100.0 ± 0.0	100.0 ± 0.0	102.5 ± 3.7	99.1 ± 4.1
overload	88.0 ± 7.4	81.3 ± 8.8	21.9 ± 1.5	17.5 ± 1.3
pacman	0.0 ± 0.0	0.0 ± 0.0	7.0 ± 0.4	3.1 ± 0.5
plants	1.3 ± 2.6	2.7 ± 3.6	23.3 ± 0.5	22.6 ± 0.5
plaqueattack	100.0 ± 0.0	93.3 ± 5.6	20.5 ± 0.7	18.5 ± 0.8
portals	77.3 ± 9.5	74.7 ± 9.8	25.3 ± 1.3	23.9 ± 1.2
roguelike	0.0 ± 0.0	2.7 ± 3.6	18.4 ± 0.5	18.4 ± 0.4
seaquest	56.0 ± 11.2	57.3 ± 11.2	24.1 ± 2.4	19.2 ± 2.3
sheriff	98.7 ± 2.6	100.0 ± 0.0	23.4 ± 0.9	21.9 ± 0.8
solarfox	6.7 ± 5.6	6.7 ± 5.6	58.1 ± 3.3	55.6 ± 3.3
superman	1.3 ± 2.6	0.0 ± 0.0	31.0 ± 1.0	30.8 ± 1.0
surround	100.0 ± 0.0	100.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
survivezombies	45.3 ± 11.3	37.3 ± 10.9	15.4 ± 1.2	11.4 ± 1.3
waitforbreakfast	60.0 ± 11.1	61.3 ± 11.0	32.7 ± 3.6	27.4 ± 2.1
whackamole	100.0 ± 0.0	98.7 ± 2.6	61.2 ± 2.5	57.9 ± 2.3
zelda	52.0 ± 11.3	77.3 ± 9.5	32.0 ± 0.6	31.6 ± 0.5
Total	53.3 ± 1.9	50.3 ± 1.9	26.8 ± 6.3	24.8 ± 6.3

Chapter 6

Conclusion

This chapter provides a conclusion of the thesis. The four research questions and the problem statement described in the first chapter are addressed based on the discussions in the other chapters. Ideas for future research are described at the end of this chapter.

6.1 Research Questions

This section discusses the four research questions described in Section 1.3 using the findings described previously.

1. *How can ideas from Iterated Width be integrated in Monte-Carlo Tree Search and enhance its performance in General Video Game Playing?*

The main idea of Iterated Width (IW) (Lipovetzky and Geffner, 2012) is the use of novelty tests for pruning states. Other than that, it simply consists of Breadth-First Search (BrFS) processes. Subsection 4.1.3 describes a new enhancement, referred to as *Novelty-Based Pruning* (NBP), which uses novelty tests for pruning in MCTS. An experimental evaluation of this enhancement shows that it provides a statistically significant increase in the overall win percentage of an MCTS agent over sixty different General Video Game Playing (GVGP) games. It has been shown to work particularly well in "puzzle" games. Additionally, *Breadth-First Tree Initialization and Safety Prepruning* (BFTI) is partially inspired by the BrFS that IW uses, and partially by the *safety prepruning* technique that Geffner and Geffner (2015) used when applying IW to GVGP. BFTI has been shown to enable an MCTS-based agent to significantly delay losses, which may improve the win percentage when combined with other enhancements in some games.

2. *How can enhancements known from other domains be used to improve the performance of Monte-Carlo Tree Search in General Video Game Playing?*

Progressive History (Nijssen and Winands, 2011) (PH), *N-Gram Selection Technique* (Tak *et al.*, 2012) (NST), *Tree Reuse* (Pepels *et al.*, 2014) (TR), and *Temporal-Difference Tree Search* (Vodopivec, 2016) (TDTS) are enhancements known from domains others than GVGP that have been described in this thesis and evaluated in GVGP. TDTS has also previously been tested in GVGP. PH and NST have been shown to provide statistically significant increases in overall win percentage when combined. This requires taking into account the locations in which actions are played when gathering statistics for actions. When this is not done, PH and NST have previously been shown not to be beneficial in GVGP (Schuster, 2015). TR has been shown to provide a statistically significant increase in overall win percentage by itself. Individually, TDTS appears to be beneficial as well in the evaluation done in this thesis, but this cannot be said with 95% confidence.

3. *How can enhancements previously proposed for General Video Game Playing be extended and used to improve the performance of Monte-Carlo Tree Search?*

In this thesis, *Knowledge-Based Evaluations* (KBE) has been discussed as an enhancement that is an extension of work previously done by Perez *et al.* (2014) and Van Eeden (2015). The basic approach is to learn which objects the avatar should or should not move towards, and combine this knowledge with distances to such objects in a heuristic evaluation function. The heuristic evaluation function used in this thesis is different from the one used by Perez *et al.* (2014) and Van Eeden (2015). Additionally, the A* (Hart *et al.*, 1968) pathfinding algorithm is used to compute distances. This was also done by Van Eeden (2015), but additional optimizations have been used in this thesis, such as A* variant 1 (Sun *et al.*, 2009). This has ultimately been shown to result in a significant increase in the overall win percentage.

4. *How can the performance of Monte-Carlo Tree Search in General Video Game Playing be improved with novel enhancements?*

The *Loss Avoidance* (LA) enhancement has been introduced in this thesis as a novel enhancement for MCTS, which has been shown to improve the overall win percentage in GVGP. This enhancement is intended to address a problem where MCTS initially obtains an overly pessimistic evaluation of certain states when they frequently lead to losses through (semi-)random play, and no longer uses more simulations to evaluate them more thoroughly.

6.2 Problem Statement

The problem statement of the thesis was formulated as follows:

How can Monte-Carlo Tree Search be enhanced to perform competitively in General Video Game Playing?

An agent that combines all the enhancements discussed for the four research questions above has been shown to significantly increase the win percentage over sixty different GVGP games. The best agent found in the experimental evaluation of this thesis includes all enhancements except for *Temporal-Difference Tree Search* (TDTS). This agent has an average win percentage of 48.4%, which is significantly better than the 31.0% of the baseline MCTS agent developed for this thesis, and the 26.5% of the Sample Open-Loop MCTS controller included in the GVG-AI framework. It is also close to, but not yet better than the 52.4% of YBCriber, which is the winning agent of the competition at the 2015 IEEE CEEC conference, and a good representation of the state of the art. Still, the MCTS agent with all the enhancements described in this thesis, except for TDTS, can be said to perform competitively in General Video Game Playing with the top tier agents. Further ideas for improving this in future research are provided in the next section.

6.3 Future Research

One way to improve the performance of the best agent found in the experimental evaluation of this thesis is to continue investigating techniques that other agents with a better performance use. An important difference between the work described in this thesis, and the best agents of 2015, is that many of those agents incorporate more domain knowledge. For instance, YBCriber gives the agent a small reward for picking up *resources*. The heuristic evaluation function of KBE could easily be extended in a similar way. This would likely improve the performance in games such as *ChipsChallenge*, which was found to be a detrimental case of NBP and KBE in Chapter 5. Many agents, such as YBCriber, also keep track of which locations of the map appear to be dangerous and correlated to losses. Such data could likely be included in the evaluation function to improve the performance in games like *Jaws*, which was described to be a detrimental case for LA in Subsection 5.2.3. Finally, the YOLOBOT agent, which won the competition at GECCO 2015, attempts to learn which areas of the map can or cannot be traversed. KBE could likely be improved by using similar techniques to take objects that block movement, or objects that result in a game loss upon collision, into account in the pathfinding algorithm used to compute distances.

A different direction of future research is to evaluate the enhancements discussed in this thesis more extensively. There are many parameters (such as the C parameter of UCB1, the novelty threshold for NBP,

W for PH, etc.) that have only been tuned slightly or chosen based on previous work (Schuster, 2015). A more extensive evaluation of different values for these parameters can likely result in a better performance. Even though *Temporal-Difference Tree Search* (TDTS) appeared to be slightly detrimental when combined with all other enhancements, it is still possible that it can be beneficial with more careful tuning of parameters. This may also involve a more extensive evaluation of *space-local normalization of value estimates* (Vodopivec, 2016).

Finally, it could be interesting to evaluate the new enhancements of this thesis in domains other than General Video Game Playing (GVGP), and in the *Two-Player Planning Track* of GVGP. *Loss Avoidance* is not expected to perform well in adversarial domains, where it would result in overly optimistic evaluations, but may be beneficial in other single-player or even cooperative domains. *Novelty-Based Pruning* is not expected to work well in adversarial games either, because it prioritizes short paths over long paths to similar states. In adversarial games, where an opponent can also make choices along the path of actions to a state, this is unlikely to be a reasonable prioritization. However, NBP may be useful in other games that are not adversarial, or even the application of MCTS in planning problems.

References

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, pp. 235–256. [12]

Baier, H. and Winands, M. H. M. (2015). MCTS-Minimax Hybrids. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 7, No. 2, pp. 167–179. [19]

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, Vol. 47, pp. 253–279. [1]

Björnsson, Y. and Finnsson, H. (2009). CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 4–15. [2]

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (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. [1, 11, 21, 24]

Campbell, M., Hoane Jr, A. J., and Hsu, F. (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1, pp. 57–83. [1]

Chaslot, G. M. J-B., Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J. W. H. M., and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [11, 24]

Childs, B. E., Brodeur, J. H., and Kocsis, L. (2008). Transpositions and Move Groups in Monte Carlo Tree Search. *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 389–395, IEEE. [21]

Chu, C. Y., Hashizume, H., Guo, Z., Harada, T., and Thawonmas, R. (2015). Combining Pathfinding Algorithm with Knowledge-based Monte-Carlo Tree Search in General Video Game Playing. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pp. 523–529, IEEE. [28]

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83, Springer Berlin Heidelberg. [1, 11]

Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T., and Togelius, J. (2013). Towards a Video Game Description Language. *Artificial and Computational Intelligence in Games* (eds. S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 85–100. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. [4]

Eeden, J. van (2015). Analysing And Improving The Knowledge-based Fast Evolutionary MCTS Algorithm. M.Sc. thesis, Utrecht University, Utrecht, the Netherlands. [28, 29, 30, 31, 32, 33, 34, 63]

Fédération Internationale de l'Automobile (2016). Points, classification and race distance. https://www.formula1.com/content/fom-website/en/championship/inside-f1/rules-regs/Classification_Race_distance_and_Points.html. Accessed: 2016-04-19. [4]

Finnsson, H. and Björnsson, Y. (2011). Game-Tree Properties and MCTS Performance. *IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11)* (eds. Y. Björnsson, N. Sturtevant, and M. Thielscher), pp. 23–30. [19]

Frydenberg, F., Andersen, K. R., Risi, S., and Togelius, J. (2015). Investigating MCTS Modifications in General Video Game Playing. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pp. 107–113, IEEE. [35]

Geffner, T. and Geffner, H. (2015). Width-Based Planning for General Video-Game Playing. *Proceedings of the Eleventh Artificial Intelligence and Interactive Digital Entertainment International Conference* (eds. A. Jhala and N. Sturtevant), pp. 23–29, AAAI Press. [15, 16, 17, 21, 23, 39, 41, 62]

Genesereth, M., Love, N., and Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI Magazine*, Vol. 26, No. 2, pp. 62–72. [1, 4]

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, Vol. 4, No. 2, pp. 100–107. [30, 32, 33, 63]

Jacobsen, E. J., Greve, R., and Togelius, J. (2014). Monte Mario: Platforming with MCTS. *Proceedings of the 2014 Conference on Genetic and Evolutionary Computation*, pp. 293–300, ACM. [35]

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [1]

Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer Berlin Heidelberg. [1, 11, 12]

Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., T. Schaul, R. Miikulainen anad, and Thompson, T. (2013). General Video Game Playing. *Artificial and Computational Intelligence in Games* (eds. S. M. Lucas, M. Mateas, M. Preuss, P. Spronck, and J. Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 77–83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. [1]

Lipovetzky, N. and Geffner, H. (2012). Width and Serialization of Classical Planning Problems. *Proceedings of the Twentieth European Conference on Artificial Intelligence (ECAI 2012)* (eds. L. De Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. Lucas), pp. 540–545, IOS Press. [2, 13, 21, 62]

Lipovetzky, N., Ramirez, M., and Geffner, H. (2015). Classical Planning with Simulators: Results on the Atari Video Games. *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 1610–1616, AAAI Press. [14]

Millington, I. and Funge, J. (2009). *Artificial Intelligence for Games*. Morgan Kaufmann, 2 edition. [44]

Nijssen, J. A. M. and Winands, M. H. M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *Lecture Notes in Computer Science*, pp. 238–249. Springer Berlin Heidelberg. [25, 62]

Pepels, T., Winands, M. H. M., and Lanctot, M. (2014). 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. [27, 28, 62]

Perez-Liebana, D., Samothrakis, S., Togelius, J., Lucas, S. M., and Schaul, T. (2016). General Video Game AI: Competition, Challenges and Opportunities. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pp. 4335–4337, AAAI Press. [2]

Perez, D. (2016). The General Video Game AI Competition. http://www.gvgai.net/. Accessed: 2016-03-24. [2, 4]

Perez, D., Samothrakis, S., and Lucas, S. (2014). Knowledge-based Fast Evolutionary MCTS for General Video Game Playing. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pp. 68–75, IEEE. [28, 29, 30, 33, 34, 63]

Perez, D., Dieskau, J., Hünermund, M., Mostaghim, S., and Lucas, S. M. (2015). Open Loop Search for General Video Game Playing. *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 337–344, ACM. [8, 27]

Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S. M., Couëtoux, A., Lee, J., Lim, C.-U, and Thompson, T. (2016). The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*. To appear. [1, 7, 28, 38, 39]

Ramanujan, R., Sabharwal, A., and Selman, B. (2010). On Adversarial Search Spaces and Sampling-Based Planning. *20th International Conference on Automated Planning and Scheduling, ICAPS 2010* (eds. R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz), pp. 242–245, AAAI. [19]

Rummery, G. A. and Niranjan, M. (1994). On-Line Q-Learning Using Connectionist Systems. Technical report, Cambridge University Engineering Department, England. [35]

Schaul, T. (2013). A Video Game Description Language for Model-based or Interactive Learning. *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pp. 193–200, IEEE Press, Niagara Falls. [4]

Schuster, T. (2015). MCTS Based Agent for General Video Games. M.Sc. thesis, Maastricht University, Maastricht, the Netherlands. [26, 27, 38, 39, 49, 62, 64]

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, Vol. 529, No. 7587, pp. 484–489. [1]

Sun, X., Yeoh, W., Chen, P., and Koenig, S. (2009). Simple Optimization Techniques for A*-Based Search. *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems*, Vol. 2, pp. 931–936, International Foundation for Autonomous Agents and Multiagent Systems. [33, 63]

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, Vol. 3, No. 1, pp. 9–44. [35]

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, Massachusetts. [26, 35, 36]

Tak, M. J. W., Winands, M. H. M., and Björnsson, Y. (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. [26, 62]

Tak, M. J. W., Winands, M. H. M., and Björnsson, Y. (2014). Decaying Simulation Strategies. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 6, No. 4, pp. 395–406. [26]

Vodopivec, T. (2016). *Monte Carlo tree search strategies*. Ph.D. thesis, University of Ljubljana, Slovenia. To appear. [35, 36, 37, 51, 62, 64]