Comparing an A-Star and a Monte Carlo Tree Search based Agent in Super Mario Bros

Bachelor Thesis

Jan Niklas Schäfer
391161

17.03.2025

Supervisor: Prof. Dr. Benjamin Blankertz
Dr.- Ing. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Kurzfassung

Diese Bachelorarbeit beschäftigt sich mit dem Vergleich zweier Mario-Agenten, basierend auf den klassischen Algorithmen Monte Carlo Tree Search (MCTS) und A*.
Beide Algorithmen wurden bereits erfolgreich zur Implementierung eines Super Mario Bros (SMB) Agenten verwendet [1][2].
Seit der ersten Implementierung der Agenten hat sich das Gebiet der automatischen Levelgenerierung stark verändert [3]. Die Anforderungen an die Agenten sind gestiegen; hier wurde untersucht, ob A* und MCTS diesen neuen Herausforderungen gewachsen wären.
Die Agenten wurden auf verschiedenen Leveltypen, älteren und neueren, verglichen und am Ende wird ein Einblick in die Skalierbarkeit der Algorithmen gegeben.
Insgesamt soll diese Arbeit einen Beitrag zum allgemeinen Fortschritt der künstlichen Intelligenz in Spielen leisten und herausfinden, inwiefern A* und MCTS in modernen Spielen anwendbar sind.
Außerdem liefert diese Arbeit eine Grundlage für zukünftige Vergleiche mit neuen Mario Agenten.

Abstract

This bachelor thesis explores a comparison between two Mario-playing agents using the classic algorithms A* and Monte-Carlo Tree Search (MCTS) in the context of Super Mario Bros (SMB). While both algorithms were implemented for SMB in the past [1][2], this thesis re implemented them in a modern framework and evaluated their performance on levels, that were not able to be generated previously.
The research aimed to determine whether these classic algorithms can be adapted to the increased complexity of modern gameplay and whether they remain relevant for artificial intelligence (ai) in gaming.
By testing the agents on a variety of levels, including both older and newer designs, this thesis seeked to provide insights into the scalability of A* and MCTS in evolving gaming environments.
The findings contribute to the broader discussion on the applicability of classic algorithms in modern ai and gaming research and will serve as a baseline for future comparison of agents within SMB.

Contents

1 Introduction **1**

2 Methods **2**
2.1 Mario-Ai-Framework . . . 2
2.1.1 Agent API . . . 2
2.1.2 Gameplay Mechanics . . . 3
2.1.3 Level Generation . . . 4
2.2 A* for SMB . . . 5
2.2.1 A* . . . 5
2.2.2 Robin Baumgarten's A* Agent . . . 5
2.3 Monte Carlo Tree Search for SMB . . . 13
2.3.1 Monte Carlo Tree Search . . . 13
2.3.2 Vanilla MCTS Agent . . . 13
2.3.3 Monte Mario Optimization . . . 16
2.4 Benchmarking Agents . . . 18
2.4.1 Level Selection and Benchmarking Sets . . . 18
2.4.2 Benchmarking Rules and Evaluation Metrics . . . 19

3 Results **21**
3.1 Comparing RBA Versions . . . 21
3.2 Comparing MCTS Agents . . . 22
3.3 Comparing A* and MCTS . . . 23

4 Discussion **26**
4.1 RBA Playstyle . . . 26
4.1.1 Original RBA . . . 26
4.1.2 RBA with Hole Detection . . . 26
4.2 Playstyle of MCTS Versions . . . 27
4.2.1 Vanilla MCTS . . . 27
4.2.2 Hole Detection . . . 27
4.2.3 Partial Selection and Roullette Wheel Selection . . . 28
4.2.4 Monte Mario . . . 30
4.3 Comparing RBA-HD and MCTS-HD . . . 30
4.4 Limitations of this Thesis . . . 31

5 Conclusion **32**

6 Utilised Ai Tools **33**
6.1 JetBrains AI . . . 33

6.2 Chat GPT 33
6.3 Deep Seek 33
6.4 DeepL 33

List of Figures

2.1 This screenshot illustrates the A* Agent of this thesis in a level. The green paths illustrate the traversed paths during the search of the previous frame. These were stored in the drawCoordinates array and subsequently displayed. It shows the agent exploring several paths that do not lead over the obstacle, the final path manages to overcome the obstacle and is subsequently chosen. 4
2.2 Illustration of a single node in the search graph with all its neighbours. The current node is marked red and all its neighbours yellow. Each possible game input in the current game state corresponds to one neighbour. ©2010 IEEE [1] 6
2.3 These figures present two plots evaluating the accuracy of the estimating functions against recorded test data. Both figures assume Mario is sprinting. 12
2.4 The image presents a visual comparison between a level generated by RPN (a) and a level generated by ORE (b). The image was taken from the meta-comparison of level generators [3]. The image illustrates the pronounced contrast in level types between a RPN and an ORE level. While this is an extreme example, it provides a useful illustration of the potential differences in the generated levels. The linearity values are part of the source material and therefore part of this figure, they can be ignored for this thesis. Further details are found in the source material [3]. ©Steve Dahlskog 2014 19

3.1 This figure presents a bar graph comparing the average final x-position of two A* agents: RBA-Orig, which refers to the RBA without Hole Detection, and RBA-HD, which represents the RBA with Hole Detection enabled. The y-axis represents the final x-position, while the x-axis labels the corresponding A* agents. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels. 22
3.2 This figure presents a bar graph comparing the average final x-position of five MCTS agents: Vanilla, which refers to the Vanilla MCTS agent, HD, which represents MCTS with Hole Detection enabled. MixMax refers to MCTS with MixMax enabled. PSRW is MCTS with Partial- and Roulette Wheel Selection activated. Monte, represents Monte Mario which has all aforementioned techniques activated. The y-axis represents the final x-position, while the x-axis labels the corresponding MCTS agents. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels. 23

3.3 This figure presents a bar graph comparing the average final x-position of the best MCTS and A* agent. RBA-HD, which is RBA with Hole Detection activated and MCTS-HD, wich is MCTS with Hole Detection activated. The y-axis represents the average final x-position, while the x-axis labels the corresponding agent. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels. 24

4.1 Two Figures representing the MCTS-Vanilla playing og-lvl-4.txt of the original and ORE benchmarking set. 28

4.2 This figure shows MCTS-PSRW playing og-lvl-4.txt of the original and ORE benchmarking set. The green lines represent the explored path of the previous frame. It can be seen that the search did not go very far beyond Mario's current position. Additionally it can be seen how a mushroom has been spawned due to the erratic behaviour of MCTS-PSRW. 29

List of Tables

3.1 This table presents the recorded data from all levels, displaying various performance metrics for each agent. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. RBA-Orig refers to the original RBA variant, while RBA-HD includes Hole Detection. MCTS-Vanilla represents the standard MCTS algorithm. MCTS-MixMax uses only the MixMax technique, and MCTS-PSRW combines Roulette Wheel Selection with Partial Selection. MCTS-HD is MCTS with Hole Detection enabled. Finally, Monte Mario integrates all the aforementioned MCTS techniques, following the original Monte Mario paper [2]. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder, as a sub folder with the same agent name [4]. 21
3.2 This table presents the recorded data from the original and ORE level set, displaying the performance metrics for the agents. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder with the corresponding agent name [4]. 25
3.3 This table presents the recorded data from the RPN level set, displaying the performance metrics for the agents. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder with the corresponding agent name [4]. 25

1 Introduction

Artificial intelligence (ai) in games has been a significant area of research for many years, with applications in gameplay agents [1] [2] and procedural level generation [3]. One of the most widely studied games in this domain is Super Mario Bros. (SMB), which has served as a benchmark for ai-driven gameplay.
Research on game playing agents gained traction with the 2009 Mario ai championship, where agents competed on randomly generated levels [1]. The most successful one was Robin Baumgarten's A*-based agent, which efficiently traversed levels using deterministic pathfinding [1]. Later, in 2014, Monte Carlo Tree Search (MCTS) was introduced as an alternative approach, demonstrating promising results for platforming ai [2].
Over time, research interest shifted from gameplay agents to procedural level generation. Since 2010, various techniques such as graph grammars [5], reinforcement learning [6], and evolutionary algorithms [7] have enabled more diverse and challenging level generation [3]. These advances have enhanced ai benchmarking by providing more varied environments for testing.
More recent research in gameplay agents has moved away from classical approaches like A* and MCTS, focusing instead on reinforcement learning-based methods [8]. This thesis aims to investigate whether A* and MCTS can still compete in more modern gameplay requirements.
To explore this, both A*- and MCTS-based agents were reimplemented within the newest Mario-Ai-Framework [9]. Their performance was then evaluated on two distinct level sets: one resembling the original competition's procedurally generated levels and another featuring more complex, part human-designed and part procedurally generated levels.
The results indicate that A* remains highly efficient and precise on simpler levels, whereas MCTS exhibits greater adaptability in complex situations. However, both approaches struggle with obstacles requiring replanning, highlighting their limitations in modern two dimensional platformers.
Ultimately, this thesis seeks to analyze the effectiveness of A* and MCTS in SMB, explore their limitations, and assess whether they remain viable for modern gameplay challenges.

2 Methods

This chapter describes how A* and MCTS can be implemented as a gameplay agent in SMB. The framework for implementation and the necessary gameplay mechanics will be described first. Then the algorithms themselves, their adaption to SMB and the techniques used to improve them will be explained. It is important to note that both algorithms face similar challenges in achieving a robust gameplay agent. The time per search is very low and the number of possible game states is very large. To adress these challenges, both algorithms have to prune their search tress in a favourable way. This chapter concludes with an introduction of the metrics and level selection for comparing the agents.

2.1 Mario-Ai-Framework

In 2006, Magnus Persson released *Infinite Mario Bros*, a Java-based version of the Super Mario Bros game [10][11]. Three years later, in 2009, Julian Togelius and Sergey Karakovskiy started the Mario ai competition. For this competition, they provided the first version of the Mario-Ai-Framework, a heavily refactored version of Infinite Mario Bros.
The two main features were random level generation and a competition API which enabled easy implementation of gameplay agents [1]. Over the years, the framework and competition underwent many iterations. Later competitions shifted the focus away from the gameplay track and focused on level generation agents [12]. This thesis focuses on the newest version of the Mario-Ai-Framework. It was released in 2019, with many improvements over previous versions. Most notably, this version comes with thousands of pre-generated levels, including a remake of the original Super Mario levels and an already implemented forward model [9].

2.1.1 Agent API

The agent API, first developed in 2009, allows for easy implementation of Mario-playing controllers. The original Super Mario Bros. runs at 25 frames per second, meaning that every 40 ms, the game state progresses one step. The agent API stops the automatic progression between frames, and the controller advances each frame itself by sending the next input to the game engine and telling it to advance its game state. In the Mario ai competition, an agent must make every decision within 40 ms, or the submission would be invalid as it would not represent a real-time playing agent [1]. The same rule applies in this thesis.
Secondly, the API allows the agent to retrieve all information on the current screen. The screen is 256x256 pixels wide and high, with the API splitting it into a 16x16 grid, where each box is 16x16 pixels. Each box contains information on all enemies, obstacles, and power-ups inside it. API calls can be modified to return only a specific type of entity, detail, or part of the grid, allowing the agent to make informed decisions about the current game state. It also allows the agent to collect metadata about the current game state, such as the game score, the remaining time in the level, or

the current mode of Mario.The agent is only allowed to make observations via this API, accessing or manipulating the memory of the game engine directly is forbidden. The code of the framework is available on GitHub [9].
In the latest iteration of the Mario-Ai-Framework(2019) a forward model is already included. A forward model enables the simulation of a game state without altering the actual game state. This is achieved by replicating the game engine, excluding the rendering component. It should be noted that there are different types of forward models that have shown promising results [13] [14]. For the purpose of consistency in comparison and simplicity in implementation, the agents will utilise the forward model provided in the 2019 Mario-Ai-Framework.
For this thesis the agent API underwent slight modifications to enable the drawing of explored paths during the current frame. Previous iterations of the Mario-Ai-Framework had already allowed for this, as demonstrated in Robin Baumgarten's popular YouTube video [15]. The latest version did not support this functionality, leading to its reimplementation. At the beginning of each frame, the agent is provided with the drawCoordinates array. This array can then be populated while the agent determines its subsequent action. Each element of the array consists of four values: the first and second values represent the x- and y-position of one game state, and the third and fourth values represent the x- and y-position of another game state. Once the agent has communicated an action to the game engine, the drawCoordinates array will be iterated over. For each entry, a line should be drawn between the first x,y coordinates (first and second entries) and the second x,y coordinates (third and fourth entries).This has been shown to be a valuable aid in understanding agent behaviour and has proven to be a highly effective debugging tool. An example frame where the drawCoordinates array was used can be found in Figure 2.1. This was the only significant alteration made to the agent API. Additionally some smaller changes to the Agent API were made, but they have no impact on the implementation of the agent or the gameplay itself and will be discussed when necessary. In the GitHub repository, all code that was previously existing and all modified code is marked accordingly [4].

2.1.2 Gameplay Mechanics

Super Mario Bros. (SMB) is a two-dimensional platform game in which the objective is to navigate the player character, Mario, through a series of levels. A level is completed by reaching the finish pole, and during the course of the game, Mario will encounter various power-ups, enemies, and obstacles. Mario will lose a level if he falls into a gap or is hit by an enemy in small mode.
SMB features three Mario modes: small, big , and fire flower. Typically Mario starts as small, picking up a mushroom power-up doubles Mario size and turns him into big Mario. Picking up a fire flower as big Mario transforms him into a fire flower, allowing him to shoot fireballs. Taking damage by touching enemies will put Mario in the previous mode, from the fire flower mode to the big mode, and from the big mode to the small mode.
The original SMB was played on a joypad with four direction buttons(up, down, left, right) and two lettered buttons (A,B). The left and right buttons control the direction of Mario, the down button lets Mario duck, the A button is used for jumping, and the B button is used for sprinting and attacking if Mario is in fire flower mode. The up arrow can be ignored as it has no effect in SMB. Each button can be pressed in combination with all other buttons, leading to a maximum of $2^5 = 32$ possible inputs per game state. Since many inputs are redundant, e.g. pressing the left and right arrows simultaneously is the same as pressing no directional button, the action space can be reduced to 12 possible inputs

Figure 2.1: This screenshot illustrates the A* Agent of this thesis in a level. The green paths illustrate the traversed paths during the search of the previous frame. These were stored in the drawCoordinates array and subsequently displayed. It shows the agent exploring several paths that do not lead over the obstacle, the final path manages to overcome the obstacle and is subsequently chosen.

without ignoring any input possibility. Additionally both agents have the down button removed, when they are small Mario, as in this case it is not needed to solve the levels. It is assumed that all agents use the full 12 possible inputs. It should be noted that there are many more game mechanics, such as different types of enemies and obstacles. These are not relevant to the implementation of the agents and will not be discussed here.

2.1.3 Level Generation

The Mario-Ai-Framework's final major feature is procedural level generation, a capability that significantly influences the gameplay experience in Super Mario Bros. Over the years, substantial research has been conducted to enhance these procedural level generators, with notable advancements beginning in 2010. Early generated levels were relatively simple, but with continued improvements, modern level generators produce more intricate levels that may even require backtracking to complete [6] [7] [16].
In 2014, a large-scale comparative evaluation of various level generators was performed, highlighting the differences in complexity and design [3]. The levels of this evaluation are incorporated into the 2019 Mario-Ai-Framework, which provides a structured approach for selecting levels for benchmarking, allowing a more informed choice when comparing the performance of different algorithms within the framework.
The selection of these levels, along with their distinctive characteristics, will be further explored in subsequent sections of this thesis.

2.2 A* for SMB

2.2.1 A*

A* is a shortest path search algorithm, first proposed in 1968 by by P. E. Hart, N. J. Nilsson, B. Raphael. It evaluates each node n using the Equation 2.1:

$$v(n) = f(n) + h(n, g) \tag{2.1}$$

where $f(n)$ is the cost to the current node n and $h(n, g)$ a heuristic estimating the remaining distance to the goal node g. This allows A* to prune the search tree and prioritize more promising nodes first. A* will find an optimal path to a goal node as long as the heuristic $h(n, g)$ is admissible and consistent. A heuristic is admissible if it never overestimates the true cost to reach the goal. A heuristic is consistent if it satisfies the triangle inequality Equation 2.2: [17]

$$h(m, n) + h(n, g) > h(m, g). \tag{2.2}$$

If the heuristic overestimates the cost by x, the final path will be no more than x too long. If only near perfection is needed, a slightly overestimating heuristic can prune the search tree and speed up the algorithm. But the margin is small, overestimating too much could worsen the performance [18].

2.2.2 Robin Baumgarten's A* Agent

This subsection discusses the implementation of Robin Baumgarten's A* agent (RBA). In 2009, Robin Baumgarten won the first iteration of the Mario ai championship with this agent. RBA was able to complete all levels of the competition in an efficient and accurate manner. In the original paper corresponding to the 2009 Mario ai championship, a rough outline of the implementation was given [1]. In addition, the official competition website provided links to Robin Baumgarten's website, where the source code was made available [19] [20]. Since the competition took place over a decade ago, many of the associated websites are now offline, including this one, and the source code was thought to be lost. Thus, an effort was made to understand and reconstruct the RBA. This section will not only discuss how the RBA can be implemented, but it will also describe the efforts made to reconstruct the agent itself.
The latest Mario-Ai-Framework included an agent called Robin Baumgarten [9]. This reference implementation, combined with the original competition paper, led to this reconstruction of the RBA. Later in the research process, the official source code of the 2009 RBA was provided personally by Robin Baumgarten, and a working version of the source code was unexpectedly found via an alternate snapshot and URL from Robin Baumgarten's website [21] [22]. Comparing the reconstruction with the original RBA source code confirms that the idea of the RBA has been transferred to this new implementation. The techniques used in RBA will be described in the following subsections.

Modelling SMB as a graph

In order to use A* to solve Mario levels, they must first be modeled as a graph. To do this, take the current displayed frame and game state as the starting node. Each possible input for that node corresponds to a neighbour, as seen in Figure 2.2. For each neighbour simulate the corresponding action, updating the game and reducing the timer by one frame. This can be repeated until all possible

game states are explored. As all levels are time-limited, the graphs will be of finite size. The graph will consist of many redundant game states, e.g. going one step to the right and one to the left will lead to the same result as waiting for two frames. This must be taken into account when implementing the A* algorithm.

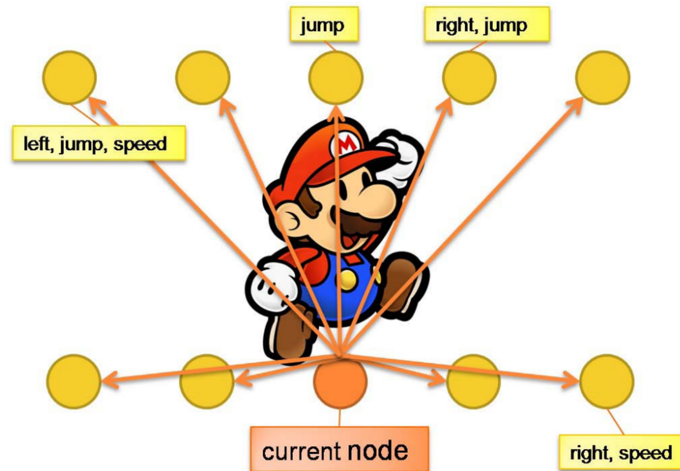


Figure 2.2: Illustration of a single node in the search graph with all its neighbours. The current node is marked red and all its neighbours yellow. Each possible game input in the current game state corresponds to one neighbour. ©2010 IEEE [1]

Adapting A* to a Time Limit

Each search cycle of the RBA is limited to 40 ms, which might not be enough time for Mario to find a winning path. To ensure Mario makes the best decisions based on his current knowledge, cache the best and furthest results during the search and return them at the end of the cycle. To do this, find the furthest position, which is the one with the biggest x-position, and the best position, which is the one with the lowest heuristic value. At the end of a search cycle, Mario should either return the best position or the furthest position, if it is a lot further than the best position. This guarantees that Mario chooses the best action based on his current knowledge.
Each search cycle lasts over two frames. This means that instead of resetting the search tree with every frame, it will be reset every second frame. This allows Mario to search with information gained during the previous frame, allowing him to search deeper in the game tree. The search state will still be reset every second frame to prevent Mario from overcommitting to a path that seems promising at first. If the path he found was previously the best, it will again be the best from the current search state.
Combining these two techniques allows RBA to find paths through the level while having only 40 milliseconds per search cycle.

Reconstructing the RBA Heuristic

The initial challenge in this research was identifying a suitable heuristic for the A* algorithm. Typically, an A* heuristic calculates the remaining distance from the current position to a goal node.

In the SMB context, this can be interpreted as the remaining x-distance required to reach the goal. However, due to Mario's forward movement being based on his current speed, this provides an inaccurate estimation of game states, as the speed is not considered in this equation. To address this challenge, the RBA employed a time-based heuristic, attempting to estimate the time remaining to reach the right border of the screen. The details of how the heuristic achieved this estimation were not properly described in the original competition paper [1]. Additionally, the heuristic found in the RBA implementation of the Mario-Ai-Framework appeared somewhat random. The subsequent section will describe the process by which this version of the heuristic was identified, utilising the reference implementation and a reverse engineering of the game engine. It should be noted that upon later access to the original RBA, it was confirmed that the current version of the heuristic closely resembled the original RBA heuristic.
The concept of time estimation is not implemented with absolute precision; rather, the heuristic does not seek to identify a precise duration remaining to achieve the goal. Instead, it assigns nodes that are able to reach the goal faster a lower value, compared to nodes that require a longer duration. Additionally, penalty factors are incorporated into the estimation, which will be discussed later. Initially, the heuristic must assign nodes capable of reaching the goal faster a lower value. This requires the calculation of the distance Mario can cover over multiple frames, with the heuristic operating under the assumption that Mario can only sprint to the right from the current game state. Two factors must be considered when calculating this. The first is the distance that Mario can cover over multiple frames if he is moving at his maximum speed, and secondly is the distance lost due to the need to accelerate to maximum speed.
Take an agent sprinting to the right on a flat level, i.e. one which does not contain gaps, enemies, nor obstacles. As the agent accelerates, two data points must be recorded for each frame: the current speed of Mario and his current x-position. These data points must then be saved in an array. Stop collecting these data points, once the speed value stops rising. This means Mario has reached his maximum speed.
Take the array of x positions and iterate over it, for each entry subtract it from the previous entry. Save this difference in a new array, each index of this new array corresponds to the forward movement in that frame. The last entry of that array will be the forward movement at maximum speed, this value will be called marioMaxMovement:

$$\text{marioMaxMovement} = 10.909058 \tag{2.3}$$

Simply multiplying this value by any number of frame will give Mario's maximum movement in that amount of frames.

$$\text{forwardMovement(frames)} = \text{marioMaxMovement} \cdot \text{frames} \tag{2.4}$$

In order determine the distance lost by accelerating to maximum speed, it is first necessary to take the previously constructed array containing the forward movement for each frame. From this construct a new array by subtracting each entry from marioMaxMovement (Equation 2.3). This gives the lostMovement array, where each entry corresponds to the lost movement compared to an agent at maximum speed.
Sum up all entries in this array. This summed up value is defined as maxForwardLoss:

$$\text{maxForwardLoss} = 88.26065980000004 \tag{2.5}$$

It corresponds to the maximum distance lost by accelerating from zero speed. This value can be used in combination with Mario's maximum speed to construct the following function:

$$\text{maxForwardLoss} = \text{marioSpeedFactor} \cdot \text{marioMaxSpeed} \tag{2.6}$$

Mario's maximum speed is the last entry of the speed array:

$$\text{marioMaxSpeed} = 9.7090845 \tag{2.7}$$

Thus marioSpeedFactor follows as:

$$\text{marioSpeedFactor} = \frac{\text{maxForwardLoss}}{\text{marioMaxSpeed}} = 9.09052339588794 \tag{2.8}$$

Rearranging the function and changing the maximum speed with the current speed of Mario then corresponds to the lost forward movement by accelerating to max speed.

$$\text{forwardLoss(speed)} = \text{maxForwardLoss} - \text{marioSpeedFactor} \cdot \text{speed} \tag{2.9}$$

The accuracy of the function can be confirmed through plotting against the testing data. The speed is taken as the x-axis, and an array is constructed where each entry corresponds to the forward movement lost from the current speed, as described in the following algorithm 1. The function can now be plotted against the constructed data sets, demonstrating a good fit.

Algorithm 1 Calculate Array of Forward Loss Corresponding to Current Speed

```
1: Initialize currentForwardLoss ← maxForwardLoss
2: for each lostMovementEntry in lostMovementArray do
3:     forwardLossArray.append(currentForwardLoss)
4:     currentForwardLoss ← currentForwardLoss − lostMovementEntry
5: end for
6: return forwardLossArray
```

Using Equation 2.4 and Equation 2.9 the following function is constructed:

$$\text{maxForwardMovement}(\text{speed}, \text{frames}) = \text{forwardMovement(frames)} - \text{forwardLoss(speed)} \tag{2.10}$$

Equation 2.10 calculates the forward movement for a fixed amount of frames and subtracts the forward loss from it. This allows the estimation of the forward movement in the next frames. The estimation is only accurate when the number of frames is big enough for Mario to accelerate to maximum speed. Otherwise Equation 2.9 will overestimate the lost distance. To avoid this problem the heuristic will always calculate with a fixed value of 1000 frames. The exact number was taken over by the reference implementation of RBA [9].

Equation 2.10 was found in an effort to understand the original heuristic. Once the equation was found and could be properly explained, it showed that the reference implementation used the same approach to calculate the forward movement. Finally the following function could be constructed:

$$\text{timeToGoal}(\text{node}) = \frac{100000 - (\text{maxForwardMovement}(\text{node.speed}, 1000) + \text{node.position})}{\text{marioMaxSpeed}} \tag{2.11}$$

Take a large constant, then calculate the maxForwardMovement and subtract it from the constant and then subtract the current x position from it. The result of this calculation is that nodes with a lower value will be in a larger x position and their maximum forward movement is greater. Thus, the function is minimised when the time to the goal is reduced. Additionally the reference implementation divided this value by marioMaxSpeed (Equation 2.7). While no explicit reasoning for it could be found, it was still taken over as it facilitated promising behaviour. It is plausible that this method was originally discovered through experimental trials.
Equation 2.11 could be used as a standalone heuristic for A*. To further improve this heuristic, two penalty factors will also be calculated: a damage penalty and a penalty if a similar node was already visited.
The equation of the damage penalty is the following:

$$\text{damagePenalty}(\text{node}) = \begin{cases} 1 \cdot (100000 - \text{size}(\text{path}) \cdot 100) & \text{if marioMode}(\text{node}) < \text{marioMode}(\text{parent}) \\ 5 \cdot (100000 - \text{size}(\text{path}) \cdot 100) & \text{if node lost the game} \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

Equation 2.12 assigns a large value to a game state if it results in the loss of the level or the taking of damage, thus maximising the value of undesirable game states. In addition, nodes that are further down the search tree are assigned a lower penalty. The values used in this function were taken from the reference implementation [9].
The equation as a whole should guarantee that, in most cases, the agent will prefer to explore a new state that did not lead to taking damage or the loss of the game, rather than further exploring game states that do so.
It is advantageous for Mario to favour game states that are further down the search tree, as this should guarantee the longest survival. Furthermore, it is possible that, due to the relatively limited time per search, the agent will be capable of identifying a path that circumvents the potential for damage or loss in subsequent frames. Thus further facilitating the need to guarantee the longest survival.
Due to the search tree containing many redundant states, the heuristic penalizes nodes that have a similar node that were already visited. This should guarantee that the agent will explore more game states, while trading off a bit of accuracy. This is achieved with Equation 2.13:

$$\text{visitedPenalty}(\text{node}) = \begin{cases} 1500 & \exists \text{ visited node where} |x_{\text{Diff}}| < 2 \text{ and} |y_{\text{Diff}}| < 2 \text{ and } |t_{\text{Diff}}| < 5 \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

$$x_{\text{Diff}} = \text{visitedXPosition} - \text{nodeXPosition} \tag{2.14}$$

$$y_{\text{Diff}} = \text{visitedYPosition} - \text{nodeYPosition} \tag{2.15}$$

$$t_{\text{Diff}} = \text{visitedNodeTime} - \text{nodeTime} \tag{2.16}$$

In order to implement the function, it is necessary to populate an array with each visited node. Then, the array must be iterated over and for each entry apply this equation. If a similar node can be found, the result will be 1500; otherwise, the result will be 0. The exact numeric value and differences that constitute a similar node were taken from the reference implementation [9].

Combining Equation 2.11, Equation 2.12 and Equation 2.13 leads to the heuristic:

$$\text{heuristic}(\text{node}) = \text{timeToGoal}(\text{node}) + \text{damagePenalty}(\text{node}) + \text{visitedPenalty}(\text{node}) \tag{2.17}$$

This heuristic enables the evaluation of previously simulated game states and the subsequent allocation of a value to them. In essence, it allows the search tree to identify promising paths and should lead a to robust SMB gameplay agent.

Estimated Simulation of Nodes

To calculate the heuristic value of a node, its associated action must be simulated. Thus each discovered state must be simulated before it can be added to the pool of possible nodes to explore next. Due to the size of the search tree and its many redundant game states, this leads to many unnecessary simulations. In addition, simulating game actions is a time-consuming task and consumes most of the computational power of the A* algorithm. Reducing the number of simulations allows the A* algorithm to explore more nodes. To achieve this, RBA introduces node estimation. Instead of simulating the associated action of a node when it is first discovered, the heuristic value of the node is estimated based on its associated action. Only when the node is then selected during the search, is its associated action simulated and its actual heuristic value calculated. If the estimated value of the node was better than it should have been, it is added back to the pool with the actual value and the next search iteration is started.
For the estimating heuristic the penalizing factors of the heuristic will be ignored. This is done to simplify the estimation process, for all game states that are not penalized the estimation should still be accurate.
It is necessary to adapt Equation 2.11 so that it can operate without an updated game state. To do so the possibility of estimating the speed gain and next position based on the next action is needed.
First construct the function that will estimate the speed gain for the subsequent frame. Take the previously constructed array of speed values of Mario sprinting in a flat level. The array is then iterated over and each element is subtracted from its previous element. The result is that this new array will contain the speed gain during each frame. The newly constructed array is then taken as the y-axis and the speed array as the x-axis. The data set was then fitted to a function, which led to:

$$\text{estimateSprintSpeedGain}(\text{speed}) = \begin{cases} 1.07 & \text{if speed} < 1.07 \\ -0.124 \times \text{speed} + 1.2 & \text{otherwise} \end{cases} \tag{2.18}$$

Repeating the same procedure for data recorded while Mario is walking in a flat level, leads to the following function:

$$\text{estimateNormalSpeedGain}(\text{speed}) = \begin{cases} 0.54 & \text{if speed} < 0.54 \\ -0.124 \times \text{speed} + 0.6 & \text{otherwise} \end{cases} \tag{2.19}$$

The accuracy of the above equations was verified by plotting the recorded data against the functions themselves, a method which demonstrated a robust estimation. Combining both equations results in the following equation:

$$\text{estimatedSpeed}(\text{node}) = \begin{cases} \text{estimateSprintSpeedGain}(\text{speed}) + \text{currentSpeed} & \text{if sprinting} \\ \text{estimateNormalSpeedGain}(\text{speed}) + \text{currentSpeed} & \text{if walking} \\ \text{currentSpeed} \cdot \frac{1}{2} & \text{if direction is changed} \end{cases} \tag{2.20}$$

Equation 2.20 assigns a value based on the current action. In the event of a change in direction in the current frame, the speed will be halved. In the actual game engine, the speed calculation while changing directions is more complex; however, halving the value allows for a good approximation. It is now possible to estimate the speed in the subsequent frame with a sufficient degree of accuracy.
The estimation of the position in the next game state is now missing. For this the function to calculate the forward movement based on the current speed is needed. Here again it needs to be differentiated between a sprinting and walking Mario. First look at Mario while sprinting. Take the previously constructed speed array as the x-axis and the forward movement array as the y-axis. Fitting a function to that data results in:

$$\text{estimateForwardMovementSprinting(node)} = \text{speed} + 1.2 \tag{2.21}$$

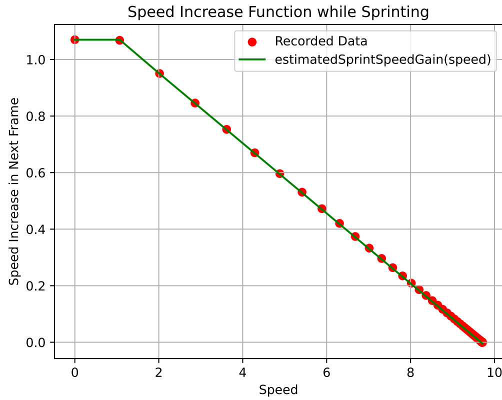
Doing the same procedure with walking Mario results in:

$$\text{estimateForwardMovementWalking(node)} = \text{speed} + 0.6 \tag{2.22}$$

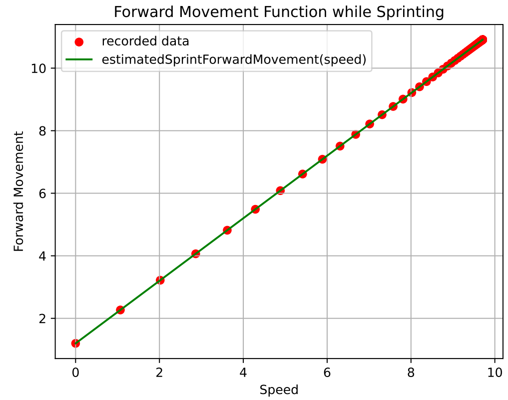
Plotting this function against the recorded data shows a well fitting function to that set. As can be seen in Figure 2.3a
If Mario is walking or sprinting to the right, simply add the corresponding value of the above equations onto Mario's current position, to estimate his next position. If Mario is walking or sprinting to the left subtract that value, as the x-position will be reduced by that value. This results in the following equation, which allows an estimation of Mario's next position based on his next action:

$$\text{estimatePosition}(\text{node}) = \begin{cases} \text{speed} + 1.2 + \text{currentPosition} & \text{if sprinting to the right} \\ \text{speed} + 0.6 + \text{currentPosition} & \text{if walking to the right} \\ -1 \cdot (\text{speed} + 1.2) + \text{currentPosition} & \text{if sprinting to the left} \\ -1 \cdot (\text{speed} + 0.6) + \text{currentPosition} & \text{if walking to the left} \end{cases} \tag{2.23}$$

Combining Equation 2.20 and Equation 2.23 now allows the heuristic to be calculated when the node is not yet simulated, leading to Equation 2.24.



(a) A plot showing the estimateSprintSpeedGain function fitted to the recorded test data. The x axis represents the current speed of Mario and the y axis the subsequent speed gain, if Mario moves in the same direction while sprinting. The green line represents the estimateSprintSpeedGain function and the red dots represent the recorded test data.



(b) A plot showing the estimatedSprintForwardMovement function fitted to the recorded test data. The x axis represents the current speed of Mario and the y axis the forward movement in that frame, if Mario moves in the same direction while sprinting. The green line represents the estimateSprintSpeedGain function and the red dots represent the recorded test data.

Figure 2.3: These figures present two plots evaluating the accuracy of the estimating functions against recorded test data. Both figures assume Mario is sprinting.

$$\text{estimatedHeuristic}(\text{node}) = \frac{\begin{matrix}100000 - \big(\text{maxForwardMovement}(\text{estimateSpeed}(\text{node}), 1000) \\ +\text{estimatePosition}(\text{node})\big)\end{matrix}}{\text{marioMaxSpeed}} \tag{2.24}$$

The collected data, fitted functions and all plots can be found on GitHub [4].

Hole Detection

Gaps pose a significant challenge during the search process, as Mario can only determine the deadly nature of a gap through the simulation of steps up to the end of the current path. To prevent Mario from mindlessly running into gaps, the Monte Mario Agent introduced hole detection, a technique that reduces the value of nodes above a gap. No further details are given in the original paper [2]. This technique showed promising results for MCTS-based agents. It was thus hypothesized that this technique would also be applicable to the RBA, and an attempt was made to adapt it to A*.
The initial versions of the hole detection implemented a penalty factor similar to the damage penalty, but this did not facilitate the desired behaviour. Consequently, a new approach for hole detection was adopted, where the nodes above a hole are not saved as either the best or furthest node. This prevents RBA from returning a node that is above a gap, and only nodes that can be simulated beyond the gap are returned. This approach addresses Mario's tendency to be short-sighted, only taking action when he is confident of avoiding a fall into a gap.
To detect if Mario is above a gap the Agent API is used. As previously described, the screen is split

into a 16x16 grid. The API enables getting the current tile position of Mario within that grid. Take the row with Mario inside and look at each tile below Mario. If all tiles are empty, Mario is above a gap and this node will not be saved as either the best or the furthest node.

The combination of all the above-described techniques results in this thesis version of the RBA, and the following pseudocode (2) shows a basic implementation of it. The implementation of this thesis is provided on GitHub [4]. Overall, this should provide a strong gameplay agent on levels that are similar to those of the 2009 competition. This is hypothesized due to the performance of the original RBA and the similarity of the implementations [1]. Subsequent chapters will not only examine this aspect, but also the performance on more modern levels.

2.3 Monte Carlo Tree Search for SMB

2.3.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm, developed in 2006 by Rémi Coulom for a Go playing agent [23]. It was first introduced as an alternative to alpha-beta search for two player zero sum games. MCTS has since then stayed relevant in game ai research, still being used in combination with modern Reinforcement Learning Techniques[24]. Although MCTS was originally developed for two player games, it can be adapted to single player games. In games with perfect knowledge it has shown promising results compared to classic search algorithms like A* [25].
A single iteration of MCTS can be broken down into four phases: Selection, Expansion, Simulation and Backpropagation.
At the start of each iteration traverse the search tree according to the selection policy. (Selection) After a node is selected, expand all its child nodes and choose the most promising node out of these children. (Expansion) Then simulate the game state of that child until a terminating game state is reached. Evaluate the terminated game state and assign a value to the child which was simulated. (Simulation) Propagate this value up the tree. (Backpropagation)

2.3.2 Vanilla MCTS Agent

In 2014, Emil Juul Jacobsen, Rasmus Greve and Julian Togelius published multiple MCTS-based SMB agents [2]. Vanilla MCTS serves as a basis for all subsequent optimisations, and will be the primary focus of this section. In the subsequent section, the optimisations of the Monte Mario agent will be presented.
The Vanilla MCTS agent functions as follows: repeat the following four steps of Selection, Expansion, Simulation and Backpropagation until a game-winning state is reached or the timer runs out.

Selection Policy

During the selection process, iterate the search tree from the root while selecting nodes according to their Upper Confidence Bound (UCB). If the selection reaches a leaf node, the expansion phase is initiated. The UCB evaluates each node using the following function:

Algorithm 2 A* Search Algorithm

```
 1: procedure A*(startingnode)
 2:     pq ← Initialize PriorityQueue
 3:     visitedNodes ← Initialize Array of Nodes
 4:     for each neighbour in Neighbours(startingNode) do
 5:         currentNode.weight ← estimatedHeuristic(node)
 6:         Add neighbour to PriorityQueue
 7:     end for
 8:     while level not won and time not run out do
 9:         currentNode ← Lowest Weight Node from PriorityQueue
10:         Simulate the associated game action of currentNode
11:         if level is lost then
12:             continue
13:         end if
14:         if heuristic(node) > estimatedHeuristic(node) + 0.1 then
15:             currentNode.weight ← heuristic(node)
16:             Add currentNode back to PriorityQueue
17:             continue
18:         end if
19:         if currentNode.visited is false and similar node in visitedNodes then
20:             currentNode.visited ← true
21:             currentNode.weight ← heuristic(node) + 1500
22:             Add currentNode back to PriorityQueue
23:             continue
24:         end if
25:         visitedNodes.add(currentNode)
26:         for each neighbour in Neighbours(currentNode) do
27:             currentNode.weight ← estimatedHeuristic(neighbour)
28:             Add neighbour to PriorityQueue
29:         end for
30:         if currentNode above a hole then
31:             continue
32:         end if
33:         if currentNode has lowest weight of all nodes yet then
34:             bestNode ← currentNode
35:         end if
36:         if currentNode is furthest node of all nodes yet then
37:             furthestNode ← currentNode
38:         end if
39:     end while
40:     if bestNode close to furthestNode then
41:         return Path to bestNode
42:     else
43:         return Path to furthestNode
44:     end if
45: end procedure
```

$$UCB_j = \overline{X}_j + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{n_j}} \tag{2.25}$$

C_p denotes the weight of the exploration term, and it was determined in the original paper that $C_p = 0.25$ yielded optimal results. This value will be used in this thesis. n_j is the number of times the node has been visited, and $\overline{X}_j$ is defined as the average reward over the number of times the node has been visited. n is defined as the total number of all expansions [2].
This UCB should guarantee a favourable balance between exploration and exploitation over many iterations.

Expansion

During expansion, generate all child nodes from the selected node. Simulate the action of each child node , select the child with the maximum reward value according to Equation 2.26.

$$reward_j = \begin{cases} 1 & \text{if game is won} \\ 0 & \text{if game lost} \\ \frac{1}{2} + \frac{1}{2} \cdot \frac{x_j - x_p}{11} & \text{otherwise} \end{cases} \tag{2.26}$$

x_j is the x-position of the current node, and x_p is the x-position of the parent node. Given that the maximum forward movement Mario can have in one frame is 10.9, it can be concluded that this reward value approaches 1 if Mario is moving to the right at maximum speed and 0 if Mario is moving at maximum speed to the left. It is noteworthy that a lost game state is the only scenario in which the reward function assigns a value of 0. Consequently, the reward function is designed to favour advancing to the left over losing the game. A similar logic is applied to the rewarding of a game-winning state with one, thereby ensuring that game-winning nodes will always be favoured.

Simulation

During the simulation process, it is first necessary to create a copy of the child that has been selected for the simulation.
Typically, MCTS simulates until a terminal game state is reached; however, in the specific case of SMB, this is not a viable option due to the significant time requirement of simulating game states. Even if it was possible, the outcome of most simulations would likely be a death due to the random nature of the process. To circumvent this issue, a rollout cap of $n_R = 6$ is employed, which limits the number of simulations carried out at each node. This value is taken over from the original Vanilla MCTS implementation[2].
Following the simulation of random actions for n_R iterations, the reward value is calculated using the following function:

$$reward_j = \begin{cases} 1 & \text{if game is won} \\ 0 & \text{if game lost} \\ \frac{1}{2} + \frac{1}{2} \cdot \frac{x_j - x_p}{11 \cdot n_R} & \text{otherwise} \end{cases} \tag{2.27}$$

This is similar to the reward function of the expansion phase, but has been adapted to account for multiple steps taken during the simulation process. This is achieved by multiplying the maximum

distance covered per frame by the rollout cap, n_R. This results in a similar behaviour to that of the expansion phase, but with the modification of multiple actions. As previously, the value would be close to one if all actions were moving to the right at maximum speed, and zero if they were moving to the left at maximum speed.

Backpropagation

After assigning a reward value to the node during simulation, the value must be propagated up the tree by iterating through all parents of the current node until the root is reached. At this point, the reward value of the simulation must be added to each node and their visit counter increased by one.

Adapting MCTS for a Time Limit

In the original Monte Mario paper, MCTS is described as an anytime algorithm, meaning it can terminate at any point and return the current best result [2].
However, the method by which this is achieved is not described. To achieve the same for all versions of MCTS, the same technique as for the RBA is employed.
After each iteration, the current node is checked to see whether it is either the furthest node or the best node. At the termination of the search, the best node is returned if it is close to the furthest node; otherwise, the furthest node is returned. This guarantees the anytime characteristic of MCTS.
In the Monte Mario paper, the vanilla MCTS agent had issues reaching a search depths bigger than five in most situations. This resulted in the agent being stuck before big obstacles or gaps [2]. Consequently, five distinct optimisations were proposed and evaluated. This thesis will concentrate exclusively on the four optimisations that enhanced the MCTS agents in accordance with the original paper [2].

2.3.3 Monte Mario Optimization

Hole Detection

As previously described hole detection was first introduced as a heuristic of the Monte Mario agent [2]. For each node, the agent detects whether Mario is above a hole. However, according to the original paper, the implementation of hole detection should result in a reduction in the reward value of nodes where Mario is above a hole. This modification also did not show any promising behaviour for the MCTS based agent, and thus the implementation was also modified to not save nodes as the furthest or best node if they are above a hole. This should prevent Mario from overcommitting to jumping over a hole without being able to simulate it until the end.
The process of detecting whether Mario is above a hole is identical to the implementation of the RBA. Take Mario's position inside the Agent API grid and look if there are any entities in the tiles below him. Allowing the agent to quickly check if Mario is above a hole.

MixMax UCB

In SMB, a considerable number of game states will have one child with a very good node value, while all other children will have a very bad node value. This results in a node having a bad average value, although one of its children will lead to a game-winning state.

One such situation is Mario jumping over a gap and being in mid-air. In this case, most actions will lead to a game-losing state and thus a bad node evaluation. Nevertheless, further exploration of this node is warranted, as all children associated with the rightward jump might lead to a game-winning path.
To circumvent this issue, a novel optimisation, MixMax UCB, was introduced to the Monte Mario agent [2]. Rather than employing the standard UCB Equation 2.25, replace $\overline{X}_j$ with following exploitation factor:

$$\text{exploitation} = Q \cdot \max + (1 - Q) \cdot \overline{X}_j \tag{2.28}$$

max is the maximum child value of the current node, and it is 0 if no child has been expanded yet. Q is the constant factor that balances the child maximum value and the actual average node value. The Monte Mario paper deducted that $Q = 0.125$ is the optimal value for this [2].
In the original Monte Mario paper, this enabled the agent to achieve a greater search depth; however, it simultaneously encouraged more reckless behaviour [2]. The optimisation itself did not result in a significant enhancement; however, in combination with subsequent optimisations, the disadvantages of MixMax UCB were attempted to be counterbalanced [2].

Partial Expansion

Simulating an action is a time consuming task, most of the time Mario will want to run or jump to the right anyways. Thus most of the simulations while expanding are unnecessary. To reduce the amount of simulations needed while exploring the search tree, Monte Mario introduced the possibility to traverse partially expanded nodes [2]. During selection rather than traversing the search tree until a leaf node is reached and expanding all its children.
Instead traverse the search tree and for each node find out if the urgency to expand a child is bigger than the UCB value of any other child, if it is, a child to expand is chosen at random.
The urgency to expand another child is calculated with the following equation:

$$UCB_c = k + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{1 + c_n}} \tag{2.29}$$

Equation 2.29 is quite similar to Equation 2.25, with the difference that nodes that are evaluated in this function do not have a value assigned to them yet. k represents the standard value of non expanded nodes, it will be set to 0.5. C_p is the exploitation constant, the same as in Equation 2.25. c_n represents the number of expanded children of the current node. n is the number of overall expansions.
In the original implementation, Partial Expansion increased the search depths of MCTS significantly in certain game situations. It helped Mario overcome large gaps and choose safer routes. At the same time it only increased search depths if one action was obviously better than others, e.g. jumping over a gap. Mario would still get stuck in front of obstacles, as jumping over them is not immediately rewarded. In these cases the exploration evened out the search tree [2].

Roulette Wheel Selection

Choosing a node to expand while being on a partially expanded node is usually done at random. Utilising domain knowledge, it is known that in most situations sprinting or jumping to the right will be

the correct action. Associating a weight with each action will allow the agent to chose those actions more often than others, this should skew the search tree in a favourable direction.
The actual weights used in the original Monte Mario implementation were not shared. The weights were originally found by observing Mario's gameplay [2]. Therefore, it was not possible to know what the original implementation looked like, and the following way of implementing this technique was tried.
To guarantee weighted random selection, take an array of possible actions. Iterate over each element of the possible actions and, if the action is sprinting to the right or jumping while sprinting to the right add it 10 times to the array of possible actions.
If the action is walking or jumping while to the right add it 5 times to the array of possible actions. All other possible actions will be added one time. Now using the Random library of Java generate a random float number between 0 to 1. Multiple that number by the size of the possible actions array and floor this value. This returns a random index of the possible actions array and allows for weighted selection of Nodes.
It should be mentioned that Roulette Wheel Selection only works in combination with Partial Expansion.

The original Monte Mario agent utilised the above-described techniques to achieve its strongest version. MixMax, UCB, Partial Expansion and Roulette Wheel selection increased search depths and skewed the search tree towards Mario going right, while hole detection circumvented Mario from mindlessly running into gaps [2]. This thesis will explore whether this still holds up in more modern levels.

2.4 Benchmarking Agents

2.4.1 Level Selection and Benchmarking Sets

Level selection is a crucial aspect of benchmarking, as different level generators produce varying types of levels. A study conducted in 2014 compared seven different platform level generators and established ways to categorize level design [3]. Based on these categorizations, two distinct sets of benchmark levels were defined.
The first set consists of levels similar to those used in the Gameplay Track competitions of 2009 and 2010. The competitions used a parametrized version of the Notch level generator with a fixed seed [1]. The Mario-Ai-Framework includes 1,000 pre-generated levels created using the random parametrized Notch (RPN) level generator, which randomizes parameters with each level generation [9]. From this set, select a random subset of 50 levels for benchmarking. These 50 levels can be found in the folder: benchmarkingLevels/randomParamNotchSet [4]. While these levels were used in past competitions, a visual inspection confirms that they are computer-generated and noticeably different from the original SMB levels. This observation is also supported by the metrics used to compare the level generators [3].
To analyze agent behaviour in levels more similar to those humans typically play, construct a second benchmarking set. The latest Mario-Ai-Framework includes a reconstruction of 15 original SMB levels. According to the 2014 study, the levels generated by the Occupancy-Regulated-Expression (ORE) level generator most closely resemble the original SMB levels. The Mario-Ai-Framework also includes 1,000 pre-generated levels using the ORE generator [9]. From this set, select 35 levels with

a fixed random seed for benchmarking. These can be found in the folder : benchmarkingLevels/originalLevelSet [4]. A visual comparison between the level types can be found in Figure 2.4.
In total, the benchmarking consists of 100 levels divided into two disjoint sets: one representing the spirit of the original Mario ai competitions and the other representing levels similar to the original SMB. The first set of test levels evaluates the performance of the agent within the new framework, while the second set assesses the performance of the agents in levels that resemble those typically played by human players.

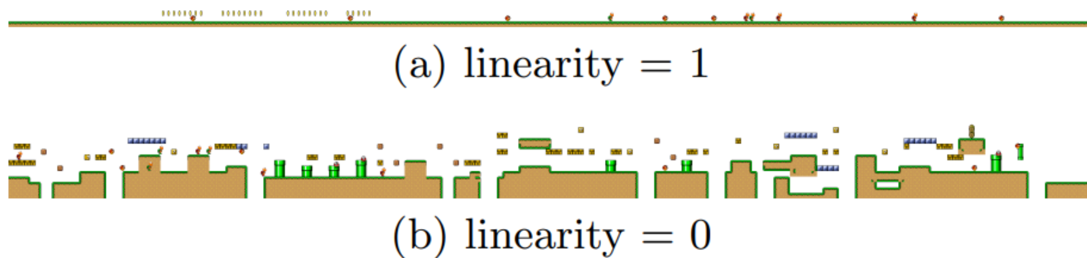


Figure 2.4: The image presents a visual comparison between a level generated by RPN (a) and a level generated by ORE (b). The image was taken from the meta-comparison of level generators [3]. The image illustrates the pronounced contrast in level types between a RPN and an ORE level. While this is an extreme example, it provides a useful illustration of the potential differences in the generated levels. The linearity values are part of the source material and therefore part of this figure, they can be ignored for this thesis. Further details are found in the source material [3]. ©Steve Dahlskog 2014

2.4.2 Benchmarking Rules and Evaluation Metrics

During the benchmarking, each level starts with Mario in his small form. The time limit per level is set to 60 seconds. Due to the random nature of the simulation of the MCTS agent and the different scheduling of the agent thread, the benchmark for all agents will be run five times and be averaged out over those results, to account for variance in results. Additionally, hardware has quite a noticeable impact on performance. To ensure the consistency of the results, the following hardware was used for all benchmarks : Intel I5-13600KF@3.5Ghz, 32GB DDR5 Ram @4800MT/s.
The validity of a benchmark is dependent on the agent's decisions being made within an average time frame of 40 milliseconds, as outlined in the original competition rules [1].
The Mario-Ai-Framework engine incorporates a 40 ms timer in each frame, and following each frame, the engine assesses if the agent's response took longer than 40 ms. If this is the case, the engine will save the lost time. In the event that the agent returns an action with time remaining on the timer, this value is to be subtracted from the lost time. If the lost time at the end of the level is found to be 0 or below 0, the benchmark is deemed valid.
The primary metric used to compare agents is the average x-position traveled across all levels. This metric provides a clear indication of the overall performance of the agent, as a higher x-position

value indicates that the agent was able to navigate further within the levels, demonstrating stronger gameplay. This metric was also used to determine the winner of the gameplay track in the Mario ai competition [1]. Each level in the RPN level set is 3120 x positions long. The level lengths for the original and ORE levels differ slightly, being on average 3081 x positions. Thus, the original and ORE levels are approximately 1.2% shorter than the RPN level set. This difference must be taken into account when comparing results between the two sets. Small differences in performance may not be meaningful and only significant differences should be used to identify trends.
First the RBA agent with and without hole detection will be compared on this primary metric.
Then the best version of the MCTS agent will be determined based on this primary metric.
In order to remain within the scope of this thesis, it is necessary to compare only the best RBA agent with the best MCTS agent. They will also be compared on this primary metric.
In addition to the primary metric, a number of secondary metrics are used to compare A* and MCTS. These are based on the original tie-breakers of the Mario ai competition [1].
The remaining time at the end of the level serves as an indicator of efficiency and aggression, as agents that complete levels quickly may prioritize speed and risk-taking behaviours. It is important to note that this metric only works as a fair comparison if the agents have a similar average final x-positions. E.g. an agent that immediately loses each level would still have a high remaining time, even though this does not necessarily indicate higher efficiency.
The number of enemies defeated is used to assess whether an agent exhibits an aggressive or passive playstyle.
Finally, the sum of Mario's mode at the end of each level provides insight into whether the agent collects power-ups.

The Mario-Ai-Framework enables the access of these metrics. With the exception of the x-position, all metrics were implemented in the MarioResult class. Furthermore, the engine had already checked if Mario took an average of 40 milliseconds per search, but only provided information via print statements if this time exceeded the threshold. The functionality to save these values was added to the framework [4][9].

3 Results

This chapter presents the results of the benchmarking runs. All agent versions made decisions within the 40 ms time limit, ensuring the validity of all reported results. A complete list of the recorded data on all levels for each agent variant can be found in Table 3.1. The following sections provide a structured presentation of the results.

Agent Version	Final X Position	Remaining Time in ms	Sum killed Enemies	Sum Mario Mode
RBA-Orig	2545.35	52192.62	236.8	0
RBA-HD	2795.28	51109.8	248.8	0
MCTS-Vanilla	2695.76	49629.72	258.4	0
MCTS-HD	2867.95	49149.24	277.2	0
MCTS-MixMax	2630.16	49572.24	250.8	0
MCTS-PSRW	356.52	43547.4	104.4	11.4
Monte Mario	294.31	42308.4	87.8	10.6

Table 3.1: This table presents the recorded data from all levels, displaying various performance metrics for each agent. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. RBA-Orig refers to the original RBA variant, while RBA-HD includes Hole Detection. MCTS-Vanilla represents the standard MCTS algorithm. MCTS-MixMax uses only the MixMax technique, and MCTS-PSRW combines Roulette Wheel Selection with Partial Selection. MCTS-HD is MCTS with Hole Detection enabled. Finally, Monte Mario integrates all the aforementioned MCTS techniques, following the original Monte Mario paper [2]. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder, as a sub folder with the same agent name [4].

3.1 Comparing RBA Versions

On all levels, the original RBA achieved an average final x-position of 2545.35. The difference in performance between the two level sets was only 0.6%. With 2562.90 for the RPN level set and 2527.80 for the original and ORE level set.

The RBA with Hole Detection (RBA-HD) achieved a final x-position of 2795.28, representing an overall 9.8% performance increase compared to the original RBA. As shown in Figure 3.1, performance on the Notch-generated levels was significantly better with an average final x position of 2901.15. Where Mario only reached an average final x position of 2689.40 on the original and ORE levels. Showing a 7.87% difference in performance on the different level sets.

Due to this improvement over the original RBA, the RBA with Hole Detection is used for further comparisons with the best MCTS version.

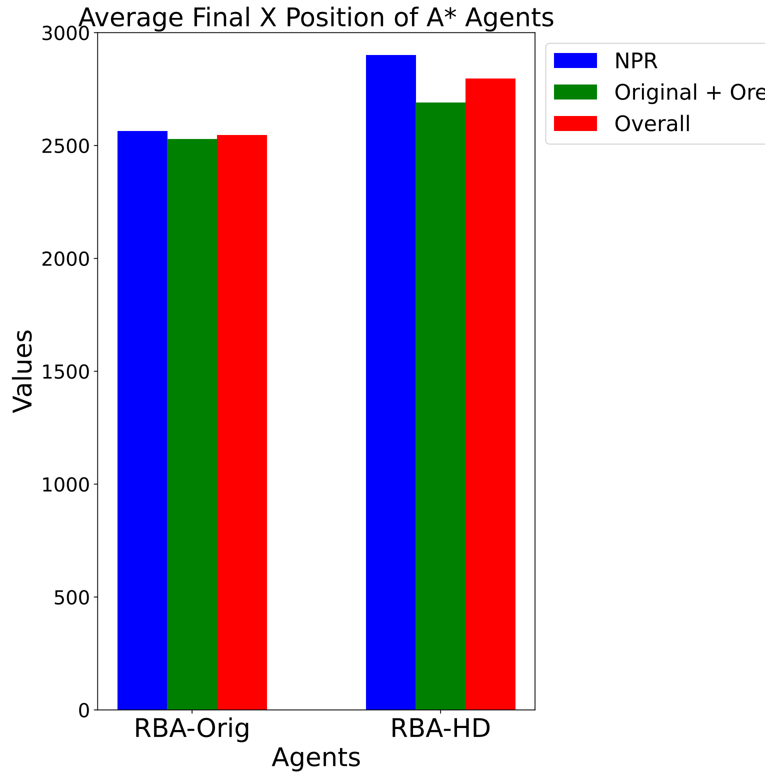


Figure 3.1: This figure presents a bar graph comparing the average final x-position of two A* agents: RBA-Orig, which refers to the RBA without Hole Detection, and RBA-HD, which represents the RBA with Hole Detection enabled. The y-axis represents the final x-position, while the x-axis labels the corresponding A* agents. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels.

3.2 Comparing MCTS Agents

Figure 3.2 presents the average final x-positions of different agent versions, showcasing their performance across two level sets: the original and ORE levels, and the RPN levels. The results highlight significant variations in performance among the agents.
MCTS-Vanilla achieved an average final x-position of 2695.76, with a 16.08% performance difference between the RPN levels (2896.41) and the original and ORE levels (2495.10).
MCTS-HD, which is MCTS with Hole Detection, performed the best overall, achieving an average x-position of 2867.95. It scored 2685.69 on the original and ORE levels and 3050.21 on the RPN levels, with a 13.57% performance difference between the two sets. Notably, MCTS-HD outperformed MCTS-Vanilla by 6.38%. Additionally one of the benchmarking runs successfully completed all 50 RPN levels.
MCTS-MixMax, MCTS with MixMax enabled, achieved an average x-position of 2630.16, performing 2.4% worse than MCTS-Vanilla. The performance difference between the RPN levels (2825.88) and the original and ORE levels (2434.44) was 16.08%.
MCTS-PSRW, MCTS with Partial Select and Roulette Wheel Selection activated, achieved an average

x-position of 356.52, which is 86.77% worse than MCTS-Vanilla. Additionally, the performance difference between the two level sets is quite drastic. The agent performed 189.19% better on the RPN levels (529.83) compared to the original and ORE levels (183.21).
The Monte Mario, MCTS with all aforementioned techniques enabled, agent performed the worst, with an average x-position of 294.31, representing an 89.08% performance loss compared to MCTS-Vanilla. Similar to MCTS-PSRW, the performance difference between the level sets is quite drastic. The agent performed 156.83% better on the RPN levels (423.67) than on the original and ORE levels (164.96).
In conclusion, MCTS-HD demonstrated the best overall performance, making it the most suitable agent for further comparison with the RBA-HD agent.

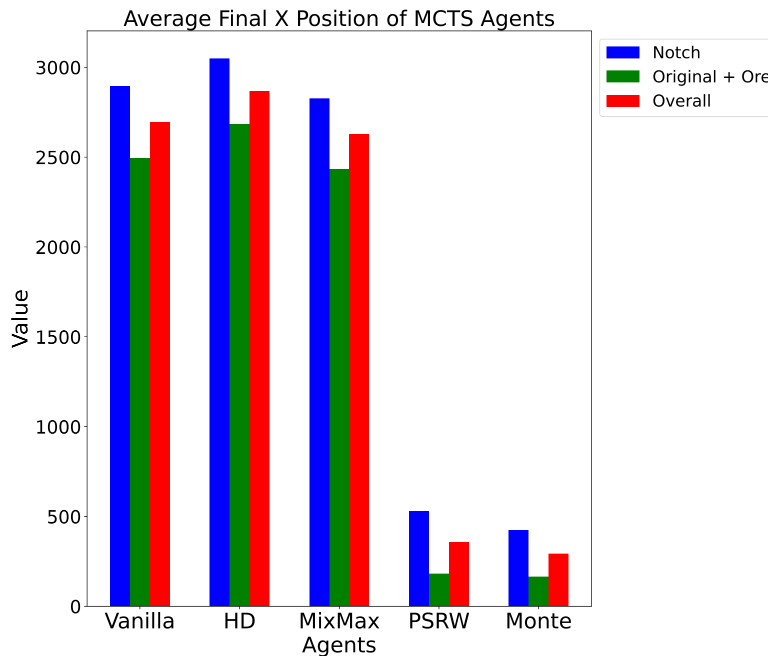


Figure 3.2: This figure presents a bar graph comparing the average final x-position of five MCTS agents: Vanilla, which refers to the Vanilla MCTS agent, HD, which represents MCTS with Hole Detection enabled. MixMax refers to MCTS with MixMax enabled. PSRW is MCTS with Partial- and Roulette Wheel Selection activated. Monte, represents Monte Mario which has all aforementioned techniques activated. The y-axis represents the final x-position, while the x-axis labels the corresponding MCTS agents. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels.

3.3 Comparing A* and MCTS

As previously mentioned, the two strongest versions of each algorithm were compared across all performance metrics: RBA-HD (A* with Hole Detection) and MCTS-HD (MCTS with Hole Detection).

The first key metric is the final average x-position. MCTS-HD outperformed RBA-HD with an average x-position of 2867.95 compared to 2795.28, representing a relative increase of 2.59%. This difference was more pronounced on the RPN level set, where MCTS-HD achieved a 5.13% higher final x-position. However, on the original and ORE level set, the performance difference was marginal only 4 x-positions in favour of RBA-HD, corresponding to a negligible 0.13% difference, in favour of RBA-HD.
Both agents failed some levels due to the in game time running out, which only occurred in the original and ORE levels. These levels sometimes required backtracking, which both agents struggled with. Figure 3.3 provides a visual comparison of the agents' performance across the different level sets.

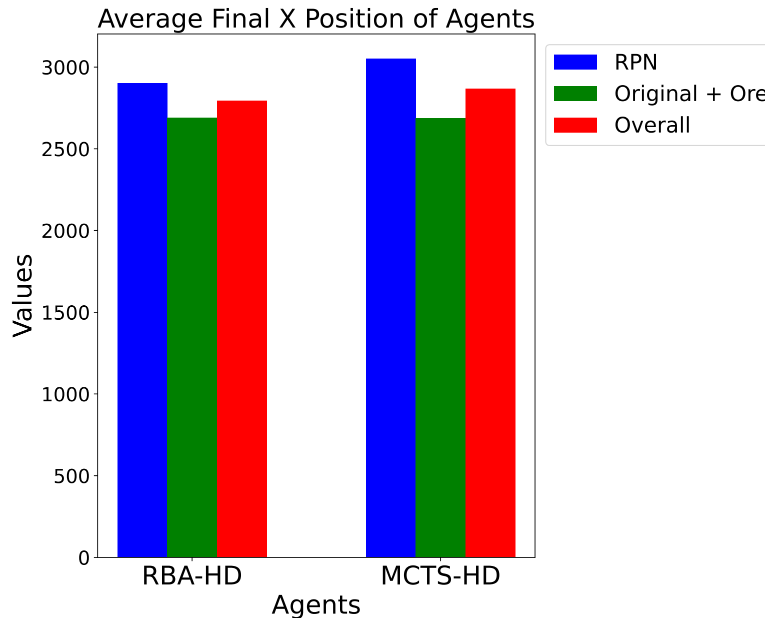


Figure 3.3: This figure presents a bar graph comparing the average final x-position of the best MCTS and A* agent. RBA-HD, which is RBA with Hole Detection activated and MCTS-HD, wich is MCTS with Hole Detection activated. The y-axis represents the average final x-position, while the x-axis labels the corresponding agent. The graph visualizes performance across different level sets, with blue bars representing results on levels generated by RPN, green bars corresponding to results on the original and ORE levels, and red bars displaying the overall performance across all levels.

Regarding the remaining time left, RBA-HD completed his benchmark slightly faster, with an average remaining time of 51,109.8 ms, compared to 49,629.72 ms for MCTS-HD 2.9% difference. Interestingly, while MCTS-HD had a consistent remaining time across both benchmarking sets, RBA-HD was about 2% faster on the RPN levels. The full benchmarking results can be found in Tables Table 3.3 and Table 3.2.
A comparison of enemy kills reveals that MCTS-HD demonstrated a more aggressive playstyle, summing up to 277.2 kills per run compared to 248.8 for RBA-HD. This value does not have to be an integer, because the kills per level were averaged out over the 5 runs and then summed up for all levels.
The difference is particularly noticeable across the level sets itself and not the agents: on the original and ORE levels, MCTS-HD achieved 203.8 kills while RBA-HD reached 183.8, whereas on the RPN

level set, the numbers dropped to 73.4 and 65, respectively.
Finally, both agents maintained a sum of Mario Mode value of zero across all levels.

Agent Version	Final X Position	Remaining Time in ms	Sum killed Enemies	Sum Mario Mode
MCTS-HD	2685.69	49184.16	203.8	0
RBA-HD	2689.40	50615.88	183.8	0

Table 3.2: This table presents the recorded data from the original and ORE level set, displaying the performance metrics for the agents. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder with the corresponding agent name [4].

Agent Version	Final X Position	Remaining Time in ms	Sum killed Enemies	Sum Mario Mode
MCTS-HD	3050.21	49114.32	73.4	0
RBA-HD	2901.15	51603.72	65	0

Table 3.3: This table presents the recorded data from the RPN level set, displaying the performance metrics for the agents. The leftmost column lists the agent versions, while the columns to the right provide their corresponding performance metrics. The recorded data corresponding to the agent can be found in the benchmarkingResults Folder with the corresponding agent name [4].

4 Discussion

This chapter presents the results of the benchmarking runs, starting with a comparison of the different RBA-based agents. Next, the performance of the MCTS-based agents is discussed. Finally, a comparison of the best performing agents highlights their strengths and weaknesses across the different level sets, while identifying the key factors influencing agent performance based on the technique used.

4.1 RBA Playstyle

The RBA agent is precise and efficient but also quite simple in its decision-making. Observing its gameplay confirms previously known issues with A*-based Mario agents [12].

4.1.1 Original RBA

RBA-Orig struggles with gaps and dead ends. It often runs off edges or gets stuck against walls, continuously trying to move right without finding a solution. This happens because the reward function mainly encourages moving to the right at high speed. When the search tree resets, the agent is again rewarded for moving right, creating a loop where it repeatedly runs into obstacles. This behaviour was already described in earlier studies [1] [12].

4.1.2 RBA with Hole Detection

Adding Hole Detection (RBA-HD) significantly reduced the number of times Mario ran off edges, improving performance by 13.19% on the notch level set. However, on the original and ORE levels, the improvement was only 6.39%. This suggests that while Hole Detection helps prevent simple mistakes, it does not solve the agent's main problem finding a successful path through more complex levels.

A major weakness of RBA-HD is its inability to replan efficiently. When Mario encounters an obstacle that requires a more complex path, he keeps moving right until no options remain. In theory, the agent could find the correct path given unlimited time, but the number of possible game states makes this unrealistic. This limitation was expected since Hole Detection only prevents falls but does not improve overall pathfinding.

To address this issue, more advanced techniques could be explored. One example is the REALM agent (2010), which utilised macro-actions to define small goals that were selected based on the current situation. The execution of these macro-actions was handled by A*, where the graph's weighting was dynamically adjusted based on the selected macro-action [26]. This adaptation ensures that the search algorithm prioritizes paths that align with the agent's current objective, allowing for more strategic decision-making while maintaining efficient pathfinding.

In summary, Hole Detection improves the RBA agent by reducing unnecessary deaths, but it does not help in levels where simple rightward movement is not enough. To compete with more advanced ai techniques, the agent would need a better way to adjust its strategy based on the level structure.

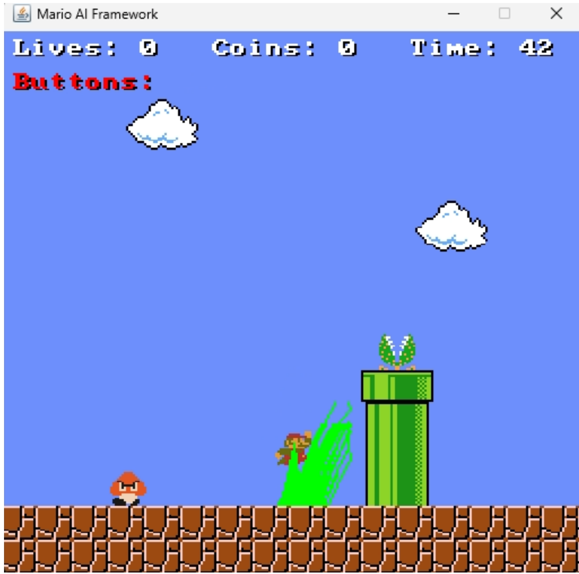
4.2 Playstyle of MCTS Versions

4.2.1 Vanilla MCTS

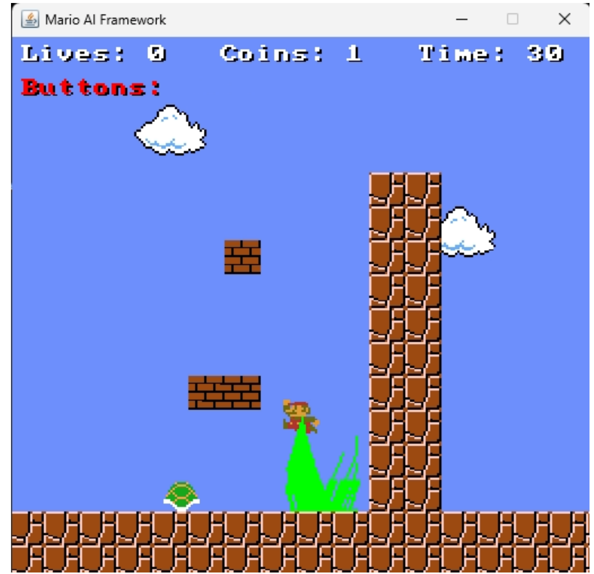
The Vanilla MCTS agent performed quite differently from the version described in the Monte Mario paper. While the original paper characterised Vanilla MCTS as cowardly and short-sighted, the version used in this paper was fast and accurate, but inconsistent [2]. The agent was always sprinting, dodging or attacking enemies at the last possible moment. However, when faced with complex obstacles such as large gaps or pipes, its behaviour became erratic.
In many cases, the agent would either run off edges or hesitate in front of pipes, taking several frames to find a valid jump. This problem stems from the reward function, which does not immediately incentivise jumping over obstacles. Instead, the agent is rewarded only after successfully clearing the obstacle, meaning that it must rely on random simulations to discover the correct path.
A significant difference between this version of Vanilla MCTS and the one in the Monte Mario paper is the search depth. The original Vanilla MCTS rarely was able to plan more than few steps ahead [2], whereas this implementation easily plans till the end of the screen . This deeper search allowed the agent to navigate obstacles more effectively. However, in situations where Mario is in front of tall structures, such as towers or large pipes, the search depth was significantly reduced. Again, this was due to the delayed reward for jumping over obstacles—Mario had to rely on randomly stumbling upon a successful solution.
The weaknesses of Vanilla MCTS are particularly evident in level `original-lvl-4.txt`, where repeated test runs led to two common failure patterns. In some cases, the agent would get stuck in front of a pipe and, after several unsuccessful jumps, end up colliding with an enemy. In other cases, the agent would reach the final obstacle of the level but fail to progress further. As can be seen in Figure 4.1a and Figure 4.1b. These outcomes highlight a major limitation of MCTS-Vanilla: its inability to effectively replan when faced with difficult level structures. The agent's reward function prioritizes moving to the right or standing still over moving left, making it difficult for the search process to explore leftward or alternative paths. Since there are too many promising game states, the agent cannot explore them all in a reasonable time frame, preventing it from effectively adjusting its strategy.

4.2.2 Hole Detection

Hole Detection had a similar effect on MCTS as it did on A*. It reduced the number of times Mario mindlessly ran off edges but did not eliminate the issue entirely.
When the agent could not find a viable jump path over a gap, it would plan up to the edge and then run off due to its momentum. In these cases, Hole Detection was unable to stop the agent from falling, as it only prevents intentional runs off edges, not unavoidable falls caused by failed planning.
Overall, Hole Detection improved MCTS performance across all level types, particularly on simpler levels where gaps were the primary challenge. This is reflected in the strong performance gains on



(a) This figure shows MCTS-Vanilla playing og-lvl-4.txt of the original and ORE benchmarking set. Mario is infront of a pipe with an enemy inside, the green lines represent the explored game states of the previous frame. It can be seen that Mario can not plan beyond the obstacle.



(b) This figure shows MCTS-Vanilla playing og-lvl-4.txt of the original and ORE benchmarking set. Mario is at the last obstacle of the level, the green lines represent the explored game states of the previous frame. It can be seen that Mario can not plan beyond the obstacle.

Figure 4.1: Two Figures representing the MCTS-Vanilla playing og-lvl-4.txt of the original and ORE benchmarking set.

RPN levels. However, in more complex levels (such as the original and ORE levels), Hole Detection could not compensate for MCTS's fundamental weaknesses in decision-making.

MixMax

MixMax was originally introduced to reduce the cautious behaviour of Vanilla MCTS by encouraging risk-taking. While it succeeded in this goal, it also led to reckless decision making, which reduced overall performance [2].
Unlike the original Vanilla MCTS described in the Monte Mario paper, the Vanilla MCTS in this thesis was not overly cautious to begin with. As a result, MixMax did not provide any noticeable improvements. Instead, it simply made the agent take more risks without solving its core weaknesses. The benchmarking results confirm this, as MCTS-MixMax performed worse than MCTS-Vanilla.

4.2.3 Partial Selection and Roullette Wheel Selection

Partial selection and roulette wheel selection are discussed together, as they are only effective when used in combination. Contrary to the original results, both techniques significantly worsened the agent's performance [2].
Partial selection was originally intended to increase search depth, but in this work it had the opposite

effect. This result is based on two key factors. First, the number of node expansions per search cycle was significantly reduced due to the computational overhead introduced by these techniques. Second, the urgency of expanding new child nodes was too high when using the values from the original paper, causing the agent to exhaust search possibilities at shallower depths before properly exploring deeper parts of the search tree.
Reducing the urgency of expanding new child nodes increased the search depth, but did not improve Mario's overall performance. The resulting paths often consisted of repetitive left and right movements, causing Mario to oscillate in place rather than progressing through the level. As shown in Figure 4.2, the algorithm failed to effectively search ahead.
Roulette wheel selection was intended to bias the search tree towards favourable actions by giving higher weights to favourable moves. However, as the original paper did not specify the exact weight values, it was assumed that right moves should be prioritised. Several weighting configurations were tested, but none produced meaningful improvements.
Although roulette wheel selection has been successful in other applications, it probably failed here because of the poor performance of partial selection. Another possibility is that the implementation of these techniques did not fully capture their intended behaviour, but given the limited details available in the original paper, they were implemented to the best of my knowledge.
Finally it should be noted, that agents with these techniques enabled were the only one with a Mario Mode sum greater than zero. This was not due to the agent finding power-ups rewarding, but rather the result of erratic movement that led to randomly picking up a power-up.

Figure 4.2: This figure shows MCTS-PSRW playing og-lvl-4.txt of the original and ORE benchmarking set. The green lines represent the explored path of the previous frame. It can be seen that the search did not go very far beyond Mario's current position. Additionally it can be seen how a mushroom has been spawned due to the erratic behaviour of MCTS-PSRW.

4.2.4 Monte Mario

Monte Mario was the best-performing MCTS variant in the original paper. It combined MixMax, Roulette Wheel Selection, Partial Selection and Hole Detection [2]. However, in this thesis, it was the worst-performing MCTS agent.
The key issue was that higher risk-taking behaviour of MixMax, combined with reduced search depth caused by Partial Selection and Roulette Wheel Selection, resulted in poor decision-making. While Hole Detection helped prevent Mario from running off edges in some cases, it could not compensate for the agent's inability to plan far enough ahead.
And the same issue observed with MCTS-HD persisted: the agent planned until the last possible pixel before a gap, but once it reached the edge, it failed to find a valid crossing path and simply ran off due to its remaining speed.
Overall it can be said the way these techniques influenced the MCTS agents in this thesis was unexpected. While Vanilla MCTS performed better than originally anticipated, it still suffers from clear limitations due to its restricted reward function. The optimizations that were supposed to improve performance did not work as intended, primarily due to search depth limitations and ineffective action prioritization.

4.3 Comparing RBA-HD and MCTS-HD

When comparing the best-performing agent versions of each algorithm, MCTS-HD outperformed RBA-HD, which contrasts with the findings of the original Monte Mario paper [2]. It is unclear which level types were used in the original Monte Mario comparison with the RBA agent. Nevertheless, this represents an improvement over the original results, even though MCTS-HD only slightly outperformed RBA-HD overall.
MCTS-HD notably outperformed RBA-HD on the RPN level set. While the exact reason for this is unclear, it is likely due to the randomness of simulation, which allows for more exploration and the discovery of game-winning paths that A* may miss. Success on these levels often hinges on slight deviations from the most promising path. However, on more complex levels, both agents performed similarly, suggesting that the increased exploration through randomness in simulation does not lead to successful solutions in more intricate levels.
Both agents perform well but struggle in similar situations due to the nature of the reward functions. Both reward functions heavily favour moving to the right and even encourage staying in the same place before moving left. While at some point the left direction will be explored all states going to the right or staying at the same place must be explored first, due to the size of the search tree this is not achievable in 40 ms.
When comparing remaining time, A* is noticeably faster and more aggressive in its approach. Visual observations further confirm this: RBA-HD moves consistently to the right, avoiding enemies and gaps in a precise and efficient manner until it suddenly can no longer find a path. In contrast, MCTS-HD tends to lose time with its occasional random movements and its tendency to get stuck in front of pipes.
Interestingly, both agents had less or similar remaining time on the original and ORE level sets compared to their remaining time on the RPN level set, despite traveling fewer x positions on the latter. This suggests that both agents struggled to be as fast and efficient on these levels as they were on the RPN levels. Further showing that these agents struggle with more complex levels.

MCTS-HD also had significantly more enemy kills than RBA-HD.
Finally, both agents had a sum of Mario Mode equal to zero, which was expected since neither agent is rewarded for picking up power-ups, meaning neither plans around them.
In conclusion, both agents display a similar playstyle across both level sets. The main limitation for both is their restricted search space, which prevents them from straying from their paths and solving more complex situations. On the other hand allowing for more exploration can quickly lead to erratic behaviour. MCTS-HD's slight randomness enabled it to explore more game states overall, though this also decreased the agent's accuracy. Filtering out bad game states using techniques like Hole Detection helped balance these inaccuracies, ultimately making MCTS-HD the strongest gameplay agent, especially on the RPN level set.
It remains an open question how MCTS could be modified to allow for a broader and more intelligent exploration of the search tree without losing its ability to go deep into it.

4.4 Limitations of this Thesis

It is important to note that certain concepts were cut from this thesis due to its limited scope. For instance, no modern approach employing reinforcement learning was implemented and compared to these classic approaches.
Additionally, a dynamic programming algorithm has been proposed as a potential solution to SMB levels, though it currently only exists in theory and has yet to be explored as a viable classic approach to address SMB levels [27]. However, the exploration of this algorithm was beyond the scope of this thesis.
Finally, it would have been possible to investigate whether the node estimation and the visited node penalty factor techniques of the RBA agent could be adapted to the MCTS agent.
These remain open questions that would require further investigation.

5 Conclusion

This thesis investigated how A* and MCTS-based agents can be implemented and whether they can perform effectively on modern levels. It was found that A* remains accurate and efficient for simple levels, but struggles with more complex scenarios where the optimal solution lies off the most promising paths. MCTS, although more adaptive, has similar weaknesses.
A surprising finding of this thesis is that the techniques that made MCTS viable for SMB in the Monte Mario paper actually worsened the performance of the MCTS agent here. In contrast, the vanilla MCTS in this paper outperformed the A*-based agent, contradicting the original findings of the Monte Mario Paper [2].
It is clear that A* works well when a rigorous heuristic can be defined, as in the simple SMB domain. In more complex environments, however, this becomes infeasible and the approach breaks down. Nevertheless, A* remains an essential tool in game ai development due to its efficiency and ease of implementation where applicable.
MCTS faces a similar challenge to A*: it requires an applicable reward function to work effectively. However, it has proven to be more flexible, for example by using reinforcement learning to evaluate game situations, rather than relying solely on domain-specific reward functions [28]. This approach could be further explored for platforming games, potentially enabling the ability to replan to the left in more complex scenarios.
It may also be worth investigating whether reinforcement learning could be used to derive a heuristic for A*. However, MCTS seems a more viable candidate given its previous results.
Another approach could involve analyzing the current game state, selecting a macro action based on that state, and adapting the reward function accordingly, as demonstrated by the REALM agent in 2010 [26]. While this method could be effective, it requires substantial domain knowledge and may not scale well to more complex games, where the number of possible macro actions could become overwhelming.
In conclusion, A* does not keep up with modern, more complex level designs, the same goes for MCTS with a domain knowledge based reward function.
Nevertheless, this research now serves as a baseline for comparison to different algorithms and can be used to evaluate whether newer approaches can achieve better gameplay performance.
Additionally MCTS, especially when combined with reinforcement learning, remains an intriguing approach for ai in games. It should be further explored how it could be used for gameplay agents, particularly in the context of three dimensional environments, where the complexity far succeeds that of a two dimensional platformer like SMB.

6 Utilised Ai Tools

This thesis utilised different ai tools for different purposes. All tools and their purposes will now be listed in this chapter. The suggestions of the following ai tools were always thoroughly reviewed and modified so they aligned with my preconceived intentions.

6.1 JetBrains AI

JetBrains AI is a tool included in the student JetBrains license granted by the TU Berlin. It was utilised as a coding assistance, not to generate complete functions.
https://www.jetbrains.com/ai/

6.2 Chat GPT

ChatGPT (GPT-4o model), developed by OpenAI, was used to help in multiple aspects. First it was used to generate latex templates and help with formatting issues. Secondly it helped in the creation of matplotlib templates and formatting of plots. Furthermore it was used to adjust grammar, spelling and expression throughout this thesis.
https://chatgpt.com/

6.3 Deep Seek

DeepSeek, was used in the same manner as Chat GPT, only as an alternative to generate a different output.
https://chat.deepseek.com/

6.4 DeepL

DeepL is an ai grammar tool, its rewrite function was used to help with spelling, expression and grammar throughout this thesis.
https://www.deepl.com/

Bibliography

[1] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[2] Emil Jacobsen, Rasmus Greve, and Julian Togelius. Monte mario: platforming with mcts. *GECCO 2014 - Proceedings of the 2014 Genetic and Evolutionary Computation Conference*, 07 2014.

[3] Steve Dahlskog, Britton Horn, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the mario ai framework. 04 2014.

[4] Jan Niklas Schäfer. Mario ai framework. https://github.com/JanNiklasSchaefer/bachelor-thesis, 2025. Accessed: 17.03.2025.

[5] E. Hauck and C. Aranha. Automatic generation of super mario levels via graph grammars. In *2020 IEEE Conference on Games (CoG)*, pages 297–304. IEEE, 2020.

[6] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[7] Ahmed Khalifa, Michael Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. pages 796–803, 07 2019.

[8] Chen Zhang, Huan Hu, Yuan Zhou, Qiyang Cao, Ruochen Liu, Wenya Wei, and Elvis S. Liu. Training interactive agent in large fps game map with rule-enhanced reinforcement learning, 2024.

[9] Ahmed Khalifa. Mario ai framework. https://github.com/amidos2006/Mario-AI-Framework, 2019. Accessed: 16.03.2025.

[10] Markus Persson. Infinite mario bros - notch's mario game. http://www.mojang.com/notch/mario/. Accessed via Web Archive: https://web.archive.org/web/20061205222530/http://www.mojang.com/notch/mario/, Accessed: 16.03.2025.

[11] IGDB. Infinite mario bros. https://www.igdb.com/games/infinite-mario-bros. Accessed: 16.03.2025.

[12] Julian Togelius, Noor Shaker, Sergey Karakovskiy, and Georgios N. Yannakakis. The mario ai championship 2009-2012. *AI Magazine*, 34(3):89–92, Sep. 2013.

[13] David Šosvald. Efficient forward model for super mario ai framework. Bachelor's thesis, Charles University, 2021.

[14] David Šosvald, Michal Töpfer, Jan Holan, Vojtěch Černý, and Jakub Gemrot. Super mario a-star agent revisited. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1008–1012, 2021.

[15] Robin Baumgarten. Infinite mario ai - long level. https://www.youtube.com/watch?v=DlkMs4ZHHr8, 2009. Accessed: 16.03.2025.

[16] Noor Shaker, Julian Togelius, Georgios N. Yannakakis, Ben Weber, Tomoyuki Shimizu, Tomonori Hashiyama, Nathan Sorenson, Philippe Pasquier, Peter Mawhorter, Glen Takahashi, Gillian Smith, and Robin Baumgarten. The 2010 mario ai championship: Level generation track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4):332–347, 2011.

[17] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[18] Ian Millington. *Artificial Intelligence for Games*. Focal Press, Philadelphia, PA, August 2005.

[19] Unknown. Related papers on mario ai. http://www.marioai.org/RelatedPapers, 2010. Accessed via Web Archive: https://web.archive.org/web/20100405100616/http://www.marioai.org/RelatedPapers, Accessed: 16.03.2025.

[20] Robin Baumgarten. Infinite super mario ai. http://www.doc.ic.ac.uk/~rb1006/projects:marioai, 2009. Accesed via Web Archive: https://web.archive.org/web/20100201180859/http://www.doc.ic.ac.uk/~rb1006/projects:marioai,Accessed:16.03.2025, Accesed: 16.03.2025.

[21] Robin Baumgarten. Infinite super mario ai. https://www.aipanic.com/h3h/dokuwiki/doku.php?id=projects:marioai, 2009. Accessed via Web Archive: https://web.archive.org/web/20150202134608/https://www.aipanic.com/h3h/dokuwiki/doku.php?id=projects:marioai, Accessed: 16.03.2025.

[22] Robin Baumgarten. Robin baumgarten a* - github copy. https://github.com/RobinB/mario-astar-robinbaumgarten/tree/master, 2013. Accessed: 16.03.2025.

[23] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[24] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023.

[25] Maarten PD Schadd, Mark HM Winands, H Jaap Van Den Herik, Guillaume MJ B Chaslot, and Jos WHM Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games: 6th International Conference, CG 2008, Beijing, China, September 29-October 1, 2008. Proceedings 6*, pages 1–12. Springer, 2008.

[26] Slawomir Bojarski and Clare Bates Congdon. Realm: A rule-based evolutionary computation agent that learns to play mario. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 83–90, 2010.

[27] Erik Domaine. Lecture 22: Dynamic programming iv: Guitar fingering, tetris, super mario bros. https://www.youtube.com/watch?v=tp4_UXaVyx8, 2011. Accessed: 16.03.2025.

[28] Tom Vodopivec, Spyridon Samothrakis, and Branko Šter. On monte carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.