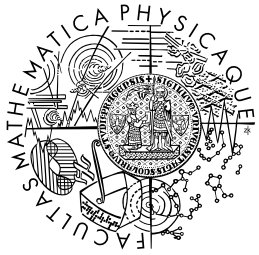


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

David Šosvald

Optimizing Super Mario game tree search

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer Science

Study branch: Visual Computing and Game Development

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D. for leading this thesis even though he already had a lot on his plate. I would also like to express gratitude to my family and friends for all their support and encouragement.

Title: Optimizing Super Mario game tree search

Author: David Šosvald

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Super Mario Bros. is still actively used as a model game for research in level generation. Every year, the most recent techniques are applied and tested. This lately includes various deep learning and reinforcement learning methods. Many of the level generators use an artificial agent to test levels' playability or to gather playthrough metrics. Therefore, the performance of the level generators is undeniably tied to the performance of the artificial agent used, both in level validation and the computing time needed.

In our previous work, we created a new state-of-the-art agent for Super Mario Bros. as a proof of concept when we implemented a more efficient forward model (world simulation) for the Mario AI framework. In this work, we continue in that work and focus on optimising how the agents explore the game tree by devising domain-specific heuristics and running extensive parameter searches to tune the agents as much as possible.

Thanks to these improvements, a new state-of-the-art agent was created. This new agent should be capable of beating every standard Super Mario Bros. level and it requires less time to solve levels than previous agents. We also present a proof of concept agent that is capable of solving maze-like levels, which is something none of the previous agents was capable of.

Keywords: artificial intelligence, game tree, domain-dependent heuristics, Super Mario

Contents

1. Introduction

This thesis follows up on our work concerning the Super Mario Bros. video game. In our previous work [1] we have revisited the Mario AI framework [2], for which we implemented a significantly better forward model. To prove the new forwards model's usefulness, we also created a new state-of-the-art Super Mario Bros. artificial agent.

We have further experimented with the artificial agents and published our work at the ICTAI 2021 conference [3], and even though the agents created were able to complete most of the Mario levels they were tested on, we still see room for improvement.

The reason why we want to make the agents better, even though they already achieved more than 98% overall win rate and over 93% win rate on the most complex level generator, is that there is still active research concerning the Super Mario Bros. game to this day, especially in the area of level generation [4][5][6][7]. All level generators need a way of testing that the generated levels can be finished – and here the artificial agents are used. The problem is that if the agent used is not performant enough, we might end up discarding levels that are perfectly playable – and possibly quite interesting since a simple agent was not able to finish them.

As we discovered during a literate review [3], many level generator authors have used an old A* agent by Robin Baumgarten (e.g. [4]), which we significantly outperformed. We believe that using more capable agents might open possibilities to generate and validate more interesting and complex levels, as such might be the hardest ones for an artificial agent to complete.

You can learn more about how artificial agents are used in scientific works concerning level generation in Chapter 7.

1.1 Problem statement

The existing agent [3] is very performant and can complete most levels flawlessly, but it is not perfect. Especially, it struggles in complex scenarios, where backtracking or choosing the correct path is required. And in our opinion, these scenarios are one of the most interesting ones. Therefore, we decided to improve our agent to be able to solve all of these situations, which will make it a perfect choice as a level validator. The concrete set of levels that we aim to solve will be discussed in Section 6.1.2. It should be noted that we decided not to deal with brick-breaking, as described in Section 6.1.3.

1.2 Goals

Our goal is to create a new state-of-the-art agent, which will be able to solve all of the levels we have available (1015 levels from 10 different generators plus 15 original Super Mario Bros. levels). We will focus on efficiently solving all of the problematic level features, such as branching and maze-like level segments. Furthermore, we want to optimize the way the agent explores the game tree, thus

making it more performant – solving the levels while exploring fewer nodes and taking less time to do so, compared to the current state-of-the-art agent [3].

1.3 Structure of the text

We start the text with two introductory chapters – Chapter 2 introduces the theoretical background of our work and Chapter 3 introduces the technical background. Following this, Chapter 4 presents an analysis of the problems that we aim to solve and introduces the various ideas that we had to solve them.

Chapter 5 documents all of our implementation work – the grid search (this is what we call a tile-level search throughout the text), the parameter searches and the new state-of-the-art agent. Chapter 6 evaluates our work and compares it with the previous results. It also selects the new best agent for the Super Mario Bros. game.

In Chapter 7, our work is put into context with previous work concerning Super Mario Bros. and artificial agents for the game, and also with work that uses the agents in the context of level generation. Chapter 8 concludes our work and presents possible future improvements and extensions.

All of the source code for this work, along with data from the experiments and instructions for using our agents and replicating our experiments can be found at `https://gitlab.com/gamedev-cuni-cz/theses/super-mario-astar`.

1.4 Copyright

Nintendo owns all rights to the Super Mario Bros. game and all its graphical assets. This thesis is not endorsed by Nintendo and uses the framework on a non-commercial basis for research purposes only. Information about the creators of the framework can be found in Chapter 7.

1.5 Acknowledgement

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Specifically, the MetaCentrum[1] computational nodes were used to run the experiments for this thesis.

[1] `https://metavo.metacentrum.cz`

2. Theory

This chapter will be a brief introduction to the theoretical background of our work. For a more detailed theoretical background, please refer to our previous work [1].

We will introduce the Super Mario Bros. game in Section 2.1, artificial agents in Section 2.2 and the A* algorithm in Section 2.3.

2.1 Super Mario Bros.

As we have already described the game in our previous work [1], we will now only focus on the aspects of the game that are important for our current work.

Super Mario Bros. is a 2D platformer game originally featuring 15 levels. Many level generators were created for the game, thus we can now have hundreds of levels of various types. But the goal for every level remains the same – to get to the finish.

The convention is that the playable character – Mario – spawns at the left border of the level and that the finish is at the right border. In its base state, the character can move left or right, it can jump and it can also sprint instead of walking to move at a greater speed. As we essentially never reach an upgraded state of the character, we do not have to deal with that.

There are three obstacles that can prevent Mario from reaching the goal. First, he may run out of time, but this basically never happens. Second, he can get killed by an enemy (by colliding with it). And third, Mario will die if he falls out of the level bounds (by falling into a pit).

As for the levels, they are made of out square blocks (tiles) and they are always 16 blocks high. Most levels are approximately 150-200 blocks wide.

Figures 2.1 and 2.2 show examples of two different Super Mario Bros. levels.

2.2 Artificial agents

This work focuses on creating and improving artificial agents for the Super Mario Bros. game. An artificial agent is a program that perceives its environment and acts upon it using a set of available actions with the intent to reach a defined goal.

In the context of Super Mario Bros., the agent perceives its environment using a forward model (described in Section 3.2), acts upon it by performing a combination of the five available actions (each action is mapped to a key, and that is either pressed or not; some actions can be combined, e.g. moving left and sprinting) and tries to reach the finish, which gets it to a game state where the level is won.

2.2.1 Artificial agents for Super Mario Bros.

The Mario AI framework that we are using runs the Super Mario Bros. game at 30 frames per second and at every frame it prompts the agent to provide the actions

Figure 2.1: A standard Super Mario Bros. level with a few different enemies shown.

that Mario will perform at the next game update. The agent receives the current state of the game (in the form of a forward model) and it has approximately 33 milliseconds to make its decision.

The agents that we are using in our work are all search-based, which means that we are using some sort of a search algorithm (A* in our case) to search through the states of the game, starting at the provided one, to find a state where the level is won. The transitions between the states (nodes of the search) are done via currently available Mario's actions. For more details, see our previous work [1], the forward model description 3.2 and the next section about the A* algorithm.

2.3 A* algorithm

The algorithm that we are using as the basis for our agents is the A* algorithm. A* is a search algorithm designed primarily for finding the shortest path between two nodes in a weighted graph. To do so, it begins at the start node and iteratively expands the most promising node available. For this, it keeps a priority queue of found nodes, and also a list of already visited nodes to avoid visiting a node twice.

Figure 2.2: An `aerial` level type featuring a lot of jumps and no solid ground.

To know which node is the most promising one, the algorithm needs a cost function that can assign a value to each node to determine how promising the node is. The cost function consists of two parts – cost of the path from the start node to the evaluated one, and a heuristic function.

The heuristic function is a function from a node to a numerical value, and it estimates the cost of the path from the evaluated node to the goal one. For the algorithm to find the optimal path, the heuristic function needs to be admissible, which means that the estimated path cost can never be overestimated (more than the lowest cost possible).

The pseudocode of the A* algorithm can be seen in Listing 2.1.

Listing 2.1: A* algorithm

```
function A*(startNode, goalNode) {
    opened.add(startNode)
    visitedCosts.init() // lowest known path costs to each node
    while (opened.size > 0) {
        current = opened.remove(); // node from opened with the
            lowest cost
        if (current == goalNode)
            return getPathToNode(current);
        for (action : getPossibleActions(current)) {
```

```
                newState = current.clone();
                newState.advance(action);
                newState.cost = pathCost + heuristic(newState);
                if (visitedCosts[newState] != unknown) {
                    if (newState.cost >= visitedCosts[newState])
                      continue;
                }
                visitedCosts[newState] = newState.cost;
                opened.add(newState);
            }
        }
        return fail; // path not found
    }
```

2.3.1 Cost function

In the standard version of A*, the cost function consists of the path cost and the heuristic function, each of them having the same weight.

But this does not need to be true. We can use a different weight for the heuristic function (or the path cost), e.g. to make the heuristic function stronger to force the search to explore states closer to the finish sooner. This version of the A* algorithm is called Weighted A* and we will be using it a lot in our work.

It needs to be noted that with the weighted version of the algorithm, we may not get optimal paths. Formally, if the cost of the optimal path is C^* and the weight of the heuristic function is W (greater than 1) while keeping the weight of the path cost at 1, we will get paths with a cost between C^* and $W \times C^*$ [8].

Also, we can include more parts in the cost function than just the two mentioned. For instance, we may have some problem-specific ratings of the nodes that we want to use to improve the search. Such extra information may be regarded as a domain-specific heuristic function and we will be using it in our work, e.g. to implement the `MFF A* Grid` agent 5.3.

3. Technological background

In this chapter, we will describe the technologies and frameworks used in our work. First, we will talk about the Mario AI framework in Section 3.1, then we will summarize the forward model that we created for the framework and used in this work in Section 3.2, and finally, we will talk about the MetaCentrum organisation whose computing infrastructure we used for our experiments in Section 3.3.

3.1 Mario AI framework

The Mario AI framework was created to facilitate the development of artificial agents and level generators for the Super Mario Bros. game. It is an open-source Java implementation of the game, which also features an API for creating custom Super Mario agents and level generators. More about the framework and the Super Mario Bros. game can be found in our previous work [1]. For the purposes of creating real-time artificial agents, the framework provides us with an important feature – a forward model.

3.2 Forward model

A forward model is essentially a simulation of the game which, given an action that we want Mario to perform, can tell us the next game state. Apart from this world update function, the forward model should also be able to efficiently clone the game state. Both these forward model functions are vital for the implementation of search-based agents, since we need a way of searching through game states.

The original forward model present in the framework was a simple wrapper of the whole game implementation, which made it quite inefficient, since it was simulating a lot of elements that are not important for an artificial agent (e.g. graphics). It was also cloning very inefficiently because it was copying all of the static level blocks, which can not change during a playthrough. The problems of the original forward model are discussed in more depth in our previous work [1].

To solve this, we previously implemented a more performant and optimized forward model that allowed us to create more efficient agents, capable of finishing more Super Mario Bros. levels in real-time. This new framework is more than 300× faster at cloning itself and takes about 20% less time on its updates that the previous one. And this is the forward model that we used for all of the agents in this work.

3.3 MetaCentrum

Our work involved searching for optimal parameters for the implemented agents. To do this, we needed to run a lot of different agent configurations on a large number of game levels and algorithm parameter configurations. As this took hundreds of days of CPU time, it would be very impractical to run these ex-

periments on our devices, therefore we utilized the services of the MetaCentrum organisation.

MetaCentrum is a Czech organisation that provides computing and storage services for researchers and students in the Czech Republic. It operates in the mode of grid computing, where individual users can dispatch their jobs to appropriate queues and the scheduling system will run them when enough resources are available.

The jobs can be generated using Bash [9] scripts, which allowed us to spawn arbitrary amounts of jobs, each with a different set of parameters. The jobs were then dispatched to many computational nodes, thus enabling them to be run in parallel.

4. Analysis

This chapter will first discuss the level segment[1] types that we identified as the hardest ones to solve in Section 4.1. Section 4.2 then talks about all of the ideas that we had to solve such segments. Some of these ideas were used in the final agent implementation, others were not, but we still believe it might be useful to talk about them and explain why we decided not to implement them.

4.1 Challenging level types

When we published our previous work [3], we also went over all of the levels that our agents did not manage to solve. We have analysed these levels to find the types of situations that are problematic for search-based agents. Let us now go over the identified segment types, so that we will know what we are dealing with in the rest of this text.

4.1.1 Branching

Most of the level segments that appear hard for a search-based agent to solve have one thing in common – they do not have an apparent linear path through them, which forces the agent to explore a big amount of game states. And if the agent fails to explore enough states to find the correct way through the level, it can end up failing by becoming stuck or falling out of level bounds.

Most of the challenging level segments could essentially be categorized as some sort of branching. See Figure 4.1 for an example. Here, the agent might start exploring the game states either in the bottom part or in the top part of the segment, but only one of these paths allows the character to progress further into the level. If the agent does not manage to calculate the correct path in the limited time period it has for its decision-making, it might end up going the bottom path and subsequently having a problem with progressing further, as it will be forced to backtrack – and backtracking can be very problematic as the agents' plans can be quite long and there is a big branching factor in the game.

Backtracking

As foreshadowed in the paragraph above, an agent might happen to initially choose the wrong path. This then means that the only way forward is to first return to a point where the level split into more possible paths, and here choose the correct one.

Backtracking may be very hard for some agent types, especially if they feature a heuristic function that makes them prefer states that are closer to the goal – which always means to the right in Super Mario Bros. This way, if they need to backtrack to the left, it is almost impossible for them due to the nodes on the path back having very large costs because of the heuristic function.

[1] We use *segment* to refer to a (usually) small part of a level.

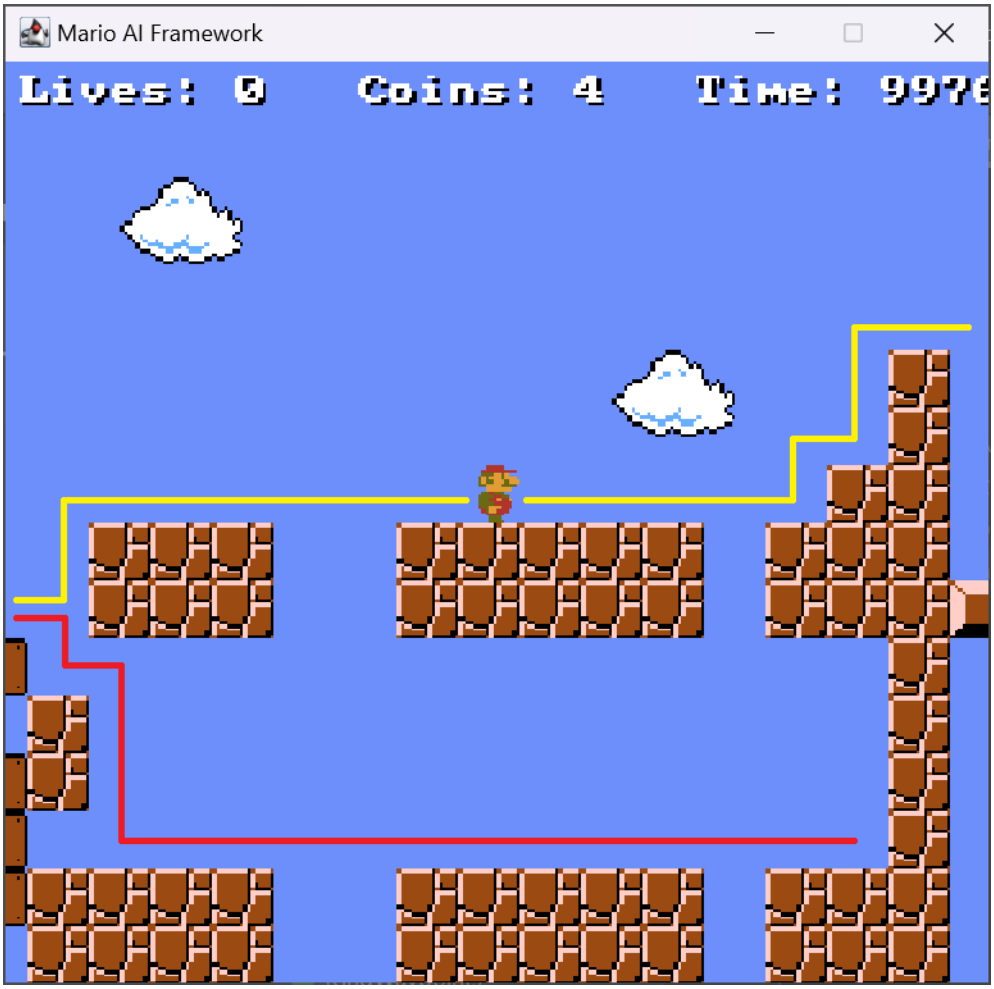


Figure 4.1: An example of a branching level segment. The correct path is marked in yellow, while the wrong path that would require backtracking is marked in red. Mario is luckily currently on the right path.

Trap pits

Some levels present in our testing set feature a special kind of branching – there is a pit present, which, if jump in by the character, is too deep to be escaped from. This means that if an agent, or a player, makes the wrong decision when choosing a path, they will no longer be able to complete the level. For an agent, this also means a huge performance hit when testing a big amount of levels, because it can not tell that it became stuck and will continue trying to find a way forward until it times out.

An example of such *trap pit*, as we decided to call it, can be seen in Figure 4.2. Here, if the agent decides to jump upwards from the left-side platform, instead of jumping down on the brick ground, it ends up trapped in a pit of solid bricks. As you can see, there is no way to get out.

4.1.2 Mazes

In our opinion, maze-like level segments or even whole maze-like levels are the most challenging feature to deal with. From some point of view, a maze-like

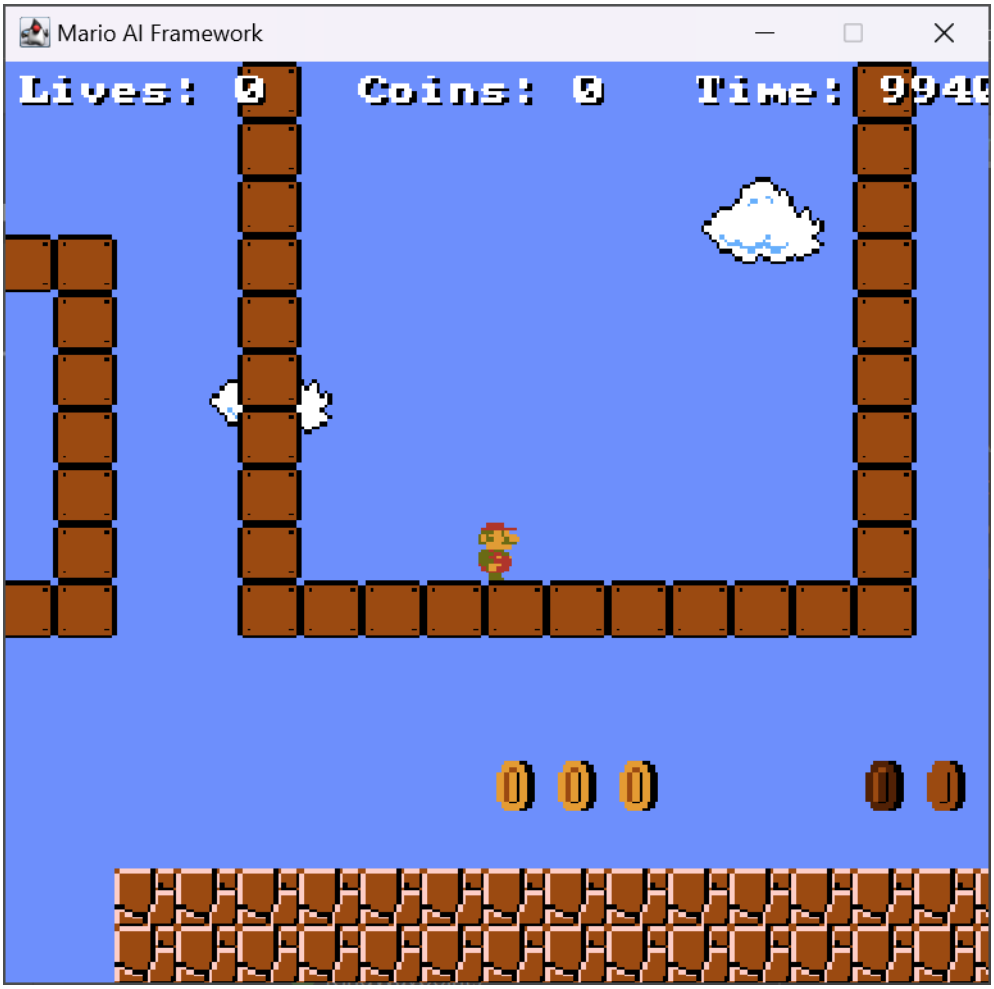


Figure 4.2: An example of a *trap pit*. If Mario falls inside, there is no way to escape. Mario, unfortunately, did just that.

segment is just a combination of branching, backtracking and dead ends, but combined in a more complex scenario.

Even the best agents from our previous work (see the `MFF-A*` agent [3]) stood no chance of solving maze-like levels. To show that we were able to solve maze-like levels, we created a special showcase level, a part of which can be seen in Figure 4.3. The whole level is a huge maze, although with a straightforward path (at least for a human player). The path through the level snakes from left to right and all the way back, alternating these directions through very long corridors (more than 100 tiles long). At the end of every corridor, we can always reach one block higher in the maze, until finally we reach the top row and can jump out of the maze right into the finish.

The catch is that most search-based agents use a goal-based heuristic, which in the case of Super Mario Bros. just tells them to always go right – towards the finish. Thus, when having to go very far to the left, all of our previous agents fail and end up hugging the right wall in the bottom row of the maze.

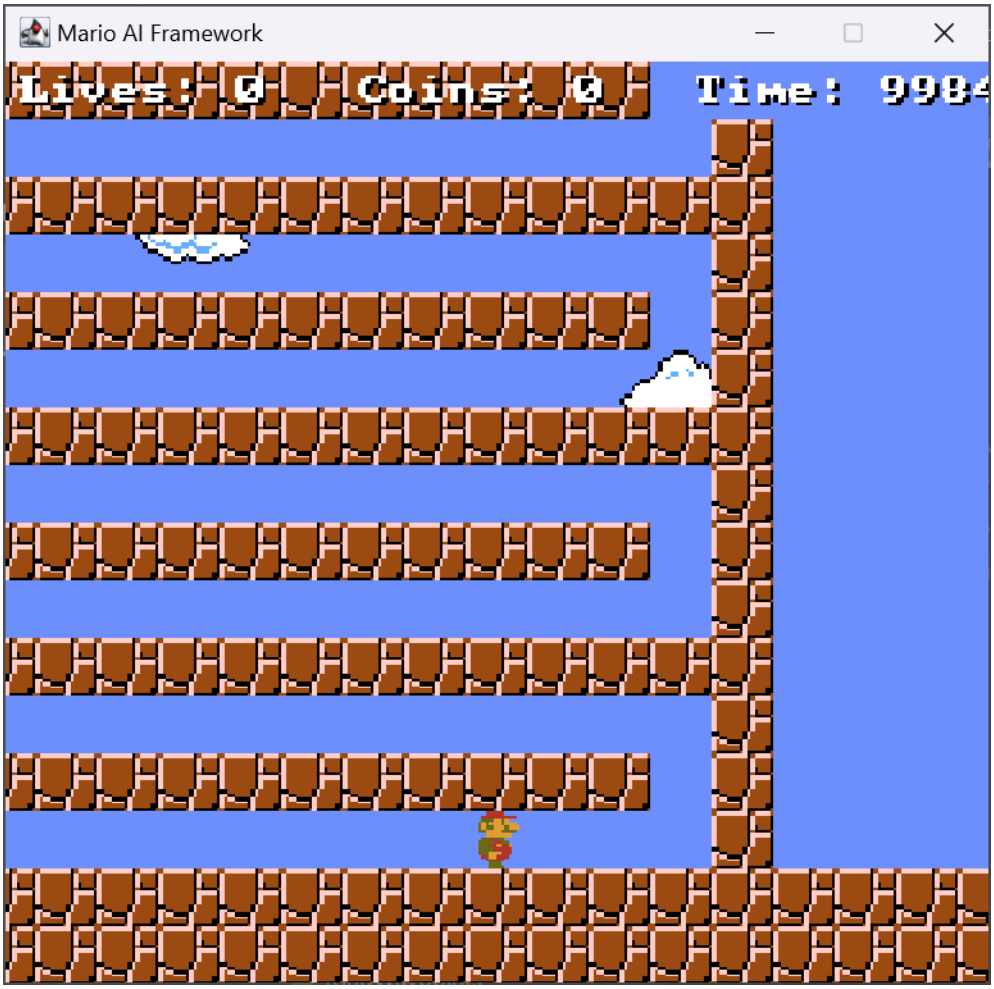


Figure 4.3: A manually created showcase level. The whole level is a huge maze and requires the agent to alternate running left and right through over 100 blocks long corridors. At the end of each corridor, the agent can ascend one floor up, until it finally gets to the top and can escape the maze and reach the finish.

4.2 Ideas

In the following section, we will talk about all of the ideas that we had for solving problems described in the previous section. They include mostly tweaks to the A* algorithm and domain-dependent heuristics. Our goal with these ideas was to create an agent capable of solving all levels, including the challenging ones that we just discussed, and to do so as effectively as possible, meaning that we wanted to optimize the exploration of the game tree. Some of these ideas were not implemented, and for those, we will explain why we decided not to implement them.

4.2.1 Parameter tuning

The cost function of the `MFF-A*` agent (the current state-of-the-art agent [3]) consists of two parts, each of which can be assigned a different weight. We believe that tuning these weights may lead to performance improvement of the agent.

Moreover, some of the agents we are planning to implement will have even more parameters that can be tweaked, thus we plan on doing extensive parameter searches in order to make the agents as performant as possible.

4.2.2 Bidirectional search

One of the first ideas we had was to use bidirectional search. Instead of a single search starting at the beginning of the level or current Mario's position, we would have two searches going on at the same time. One would start at the same place as the initial one-way one and the other would start at the finish of the level.

This improvement was meant to reduce the number of nodes that we would need to explore, since if the distance to the goal is d and the branching factor is b, a standard one-way search (without heuristic) will explore b^d nodes, but a bidirectional search only needs $b^{d/2} + b^{d/2}$ nodes, which is significantly less. And even when employing a heuristic function, this might greatly reduce the number of visited nodes. Moreover, some levels might be easier to solve when running from right to left.

However, we realised that it would be very hard and impractical to implement a backwards search for Super Mario Bros., as we would need a backward version of the forward model 3.2, which is not easy to implement.

There are three main reasons for this. The first one is that the environment of the game is dynamic (e.g. enemies moving) and therefore searching from the finish to the start would not sync well and enemies would have different positions than anticipated when executing the path from the start.

The second reason is that not all paths that Mario can take can be taken both ways. Imagine the backward search jumping down from a high wall. When we would attempt to follow this path in the opposite way, we would not be able to get up on the wall.

The final reason is that we would get a series of Mario inputs that should be taken to go from the finish to the start, but we need to reverse these inputs to go from start to finish. The problem is that the search might have calculated with some momentum when jumping over a pit, let us say when going from right to left. This means it accelerated before the pit, on its right side, then it jumped over, and finally, upon landing on the left side, it slowed down.

If we simply reversed the left and right inputs, we would slowly approach the pit from the left side and attempt to jump over it. But since we did not accelerate, we might end up not making it to the other side. To put it more generally, it would be very hard, if not impossible, to translate the actions from one direction into the other.

4.2.3 Tile constraints

Since the Super Mario Bros. game is tile-based, we came up with the idea of marking the tiles with different information. More concretely, if we had a way of knowing the approximate path or corridor leading to the finish, we could mark such tiles and try to stay on them or close to them. This might help with avoiding dead ends and stop us from exploring useless game states. And we could also use this information in two modes – as a soft or as a hard constraint.

For an idea of how to get the information about the approximate path through the level, please see the section about grid utilization 4.2.5.

Soft constraint

In the soft constraint version, we could adjust a search node's cost based on its distance from the corridor that we want the agent to follow (using the distance from the intended path as a domain-specific heuristic function). The farther the agent would be from the path, the more it would get penalized. And the distance scaling of the penalization does not need to be linear, we could experiment with many different variants of it. This way the agent's node exploration would be guided towards states that lie close to the approximate path.

This idea is one of those that ended up being implemented and it has proven extremely helpful in solving the branching issues. And since the agent could always choose the correct path, we would not have to deal with backtracking at all. The details can be found in Section 5.3.

Hard constraint

In the hard constraint version, instead of guiding the agent away from unwanted nodes by adjusting the cost function, we could simply forbid the agent from entering game states where Mario is too far away from the approximate path.

We would need to leave some manoeuvring room in the tiles close to the path to allow the agent to dodge enemies, but the tiles that would be far away from the path could be considered a dead end and completely removed from the agent's search – if a state at such tile would be opened, it would be discarded.

We did not end up implementing this idea as the soft version of the tile constraint proved to work very well and it is much more flexible.

4.2.4 Suggested moves

When the search expands a node (a state of the game), it creates new nodes by applying all of possible Mario's moves (e.g. run right, jump right, run left). If we knew which way we want to go next, we could use this to know which Mario actions to try first and thus prioritize the corresponding new nodes in the search, making the exploration of the game tree search more efficient.

For one option of how to know which actions to prioritize, please see the following section.

4.2.5 Grid utilization

We were trying to come up with a way of solving the higher-level navigation problem in the levels. We knew that the agents are basically perfect at managing local-level navigation and dodging enemies, but they can struggle in branching and maze-like scenarios. Therefore, we have decided to try utilizing the fact that the Super Mario Bros. levels are made out of tiles.

Each of the levels is 16 tiles high and usually around 200 tiles wide. We thought that we might first find a path in the level grid, marking the subsequent

tiles that an agent should follow, and then use this to guide the actual search – a similar idea is used in hierarchical path-finding.

Since there are only a few thousand tiles in a level, finding a path in this search space would be way easier than searching through the individual game states, where Mario might only move a few pixels per step, making such search space larger by several orders of magnitude.

This turned out to be the best idea we had. We used a grid-level A* algorithm to find a rough path and then utilized it to guide the agent. The implementation details can be found in Section 5.2 and a new agent based on this idea – `MFF A* Grid` – is described in Section 5.3.

From now on, we will refer to the tile-level path as a `grid path`, and to the process of obtaining it as a `grid search`.

Figure 4.4 shows how the grid path might look like for the branching level segment shown in Figure 4.1. Let us now consider how the grid path could be used to implement some of the ideas presented above.

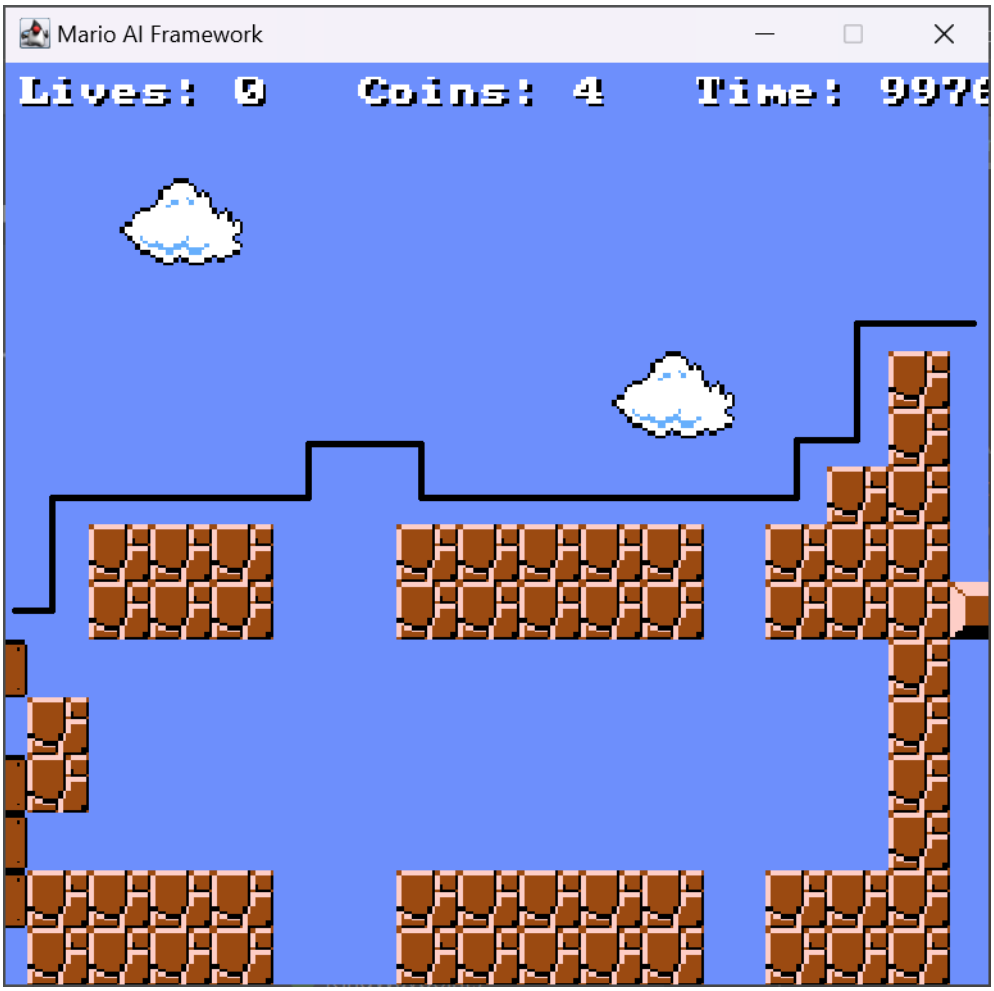


Figure 4.4: The black line visualises the grid path that our implementation of the grid search returned. Notice that it completely avoids the lower part of the level segment, because it does not lead any further.

Tile constraints implementation using the grid path

As mentioned in the description of the tile constraint idea 4.2.3, we would first need a way of getting an approximate path through the level. And since we just proposed utilizing the grid that creates the Super Maro Bros. levels to find a grid path, such path is exactly what we need, and what we used, for the implementation of tile constraints.

Suggested moves implementation using the grid path

Suppose that we now have the grid path available. That means we can now tell which moves to prioritize by looking at where the next tile of the path is and selecting the move that will get us there. Thanks to this we could implement the suggested moves idea 4.2.4.

The problem is that this would only work well when the agent occupies a tile that is a part of the found grid path. When the agent is not on the grid path, it is not clear which tile of the path it should navigate to (e.g. there could be a wall between the agent and the tile we would choose).

There might be a workaround to this, but since we had all of the levels solved before this was implemented using other ideas, we decided that this idea is no longer worth using.

4.2.6 Waypoints

The last idea, specifically targeted to solve maze-like environments 4.1.2, involves creating sub-goals throughout the levels and using them to have more control over the agent's navigation through the level.

We could utilize the grid path presented in the previous section 4.2.5 to sample the waypoints from it. We could simply create a navigation point every few blocks of the grid path and the agent would then navigate between the individual waypoints.

This would reduce the (sub)goal distances that the agent would have to travel, which may prevent it from getting stuck. We could tweak the cost function to take the current waypoint that we are trying to reach into account or to even consider more waypoints at a time. We could also experiment with the waypoint density.

This idea was used to create the `MFF A* Waypoints` agent 5.4.

5. Implementation

In this chapter, we will talk about the artificial agents that we implemented. We will start by trying to find the best parameters for the `MFF A*` agent (the best agent from our previous work [3]) in Section 5.1. Then, we will explain how we implemented the grid search in Section 5.2 and explain how we used it to create a new agent in Section 5.3. Finally, in Section 5.4 we will try to tackle maze-like environments.

The source code of the implementation can be found at `https://gitlab.com/gamedev-cuni-cz/theses/super-mario-astar`.

5.1 MFF A* parameter tuning

`MFF A*` is the best artificial agent that we created in our previous works. It was created as a part of our work at improving the forward model for Mario AI framework [1], where it is also described in more detail. The general idea is that it uses the A* algorithm to search through the states of the game, each node of its search representing a state (a tick) of the game.

As it uses the A* algorithm 2.3, it needs a cost function, which assigns costs to individual nodes of the search. Our cost function sums the depth of the node (distance from the start in ticks) and the minimal distance to the goal (given running in a straight line at maximal speed, also in ticks). The question is, what weights to assign to the node depth and the time to finish components.

In our previous work, we used a node depth weight (NDW) of 1.0 and time to finish weight (TTFW) of 1.1, but in order to reduce the search space, we also used a trick to apply each selected action three times to get a new state, while only incrementing NDW once. This means that the value of NDW was effectively $\frac{1}{3}$, making the heuristic very strong.

We wanted to revisit these parameters and use the MetaCentrum computational resources to run a parameter search to help us optimize the parameters – both for the number of levels solved and for run time.

We have decided to try values from 0.2 to 4.0 with a step of 0.2 for NDW. For TTFW, we used values from 0.2 to 2.0 with a step of 0.2. We have tried all combinations of these parameters, meaning 200 parameter sets in total, covering a big range of different parameter ratios.

We have tested each of the parameter sets on 1015 levels – 100 levels per level generator we have available and the 15 original Super Mario Bros. levels. For each parameter set, we sent a job to the MetaCentrum cluster. The evaluation of the parameter search as well as the best parameter set can be found in Section 6.2.

5.2 Grid search

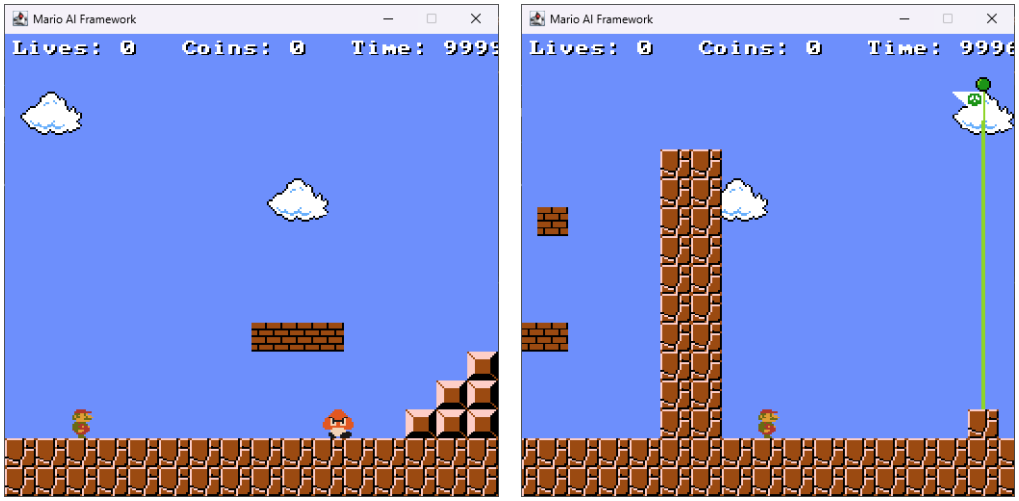
As already mentioned 4.2.5, utilizing the grid of tiles that composes the Super Mario Bros. levels was the greatest improvement we made to the search-based agents. The improvement consists of two steps – finding the grid path, and using

it to guide the agents' search. In this section, we will go over the details of implementing the grid search.

As we mentioned earlier 2.1, every Super Mario Bros. level is made of square tiles (blocks) of the same size. Some of them are solid and make up the ground and walls, some of them are empty, and some of them are special (jump-through platforms). The levels are always 16 tiles high, and generally around 200 tiles long. Mario, the character we control, starts on the left side of the level. The goal we want to reach, represented by a flag, is on the right side of the level.

Our goal here is to implement an algorithm that will receive the tiles of a given level and mark the tile-level path from the start (Mario's initial position) to the goal flag. For illustrations of a start position, a goal flag and a segment of a level, please see Figures 5.1 and 5.2.

The correctness of the grid search is tested in Section 6.3.



(a) The start of a level with Mario's spawn position,

(b) The end of a level featuring a goal flag.

Figure 5.1: The start and the end of level 4 from the `original` level pack.

5.2.1 A* for a platformer game

The way we are going to implement the grid search is by using the A* algorithm 2.3, but we need to solve the fact that Super Mario Bros. is a platformer game that, as opposed to a top-down game, does not allow deliberate movement in all directions. We can not always go up, because to move up we need to perform a jump and to do so, we either need to be standing on the ground or already performing a jump and not having reached its peak yet. It is also not always possible to move down, as most of the blocks we might be standing on are solid.

We have considered solving the problems with vertical movement using a `jump force` method, where we would remember how many times can we move upwards before having to start falling down, but we soon realized that Mario's movement will require a more complex solution. Therefore, let us first talk in detail about how Mario's movement works in the game before we get to the implementation itself.

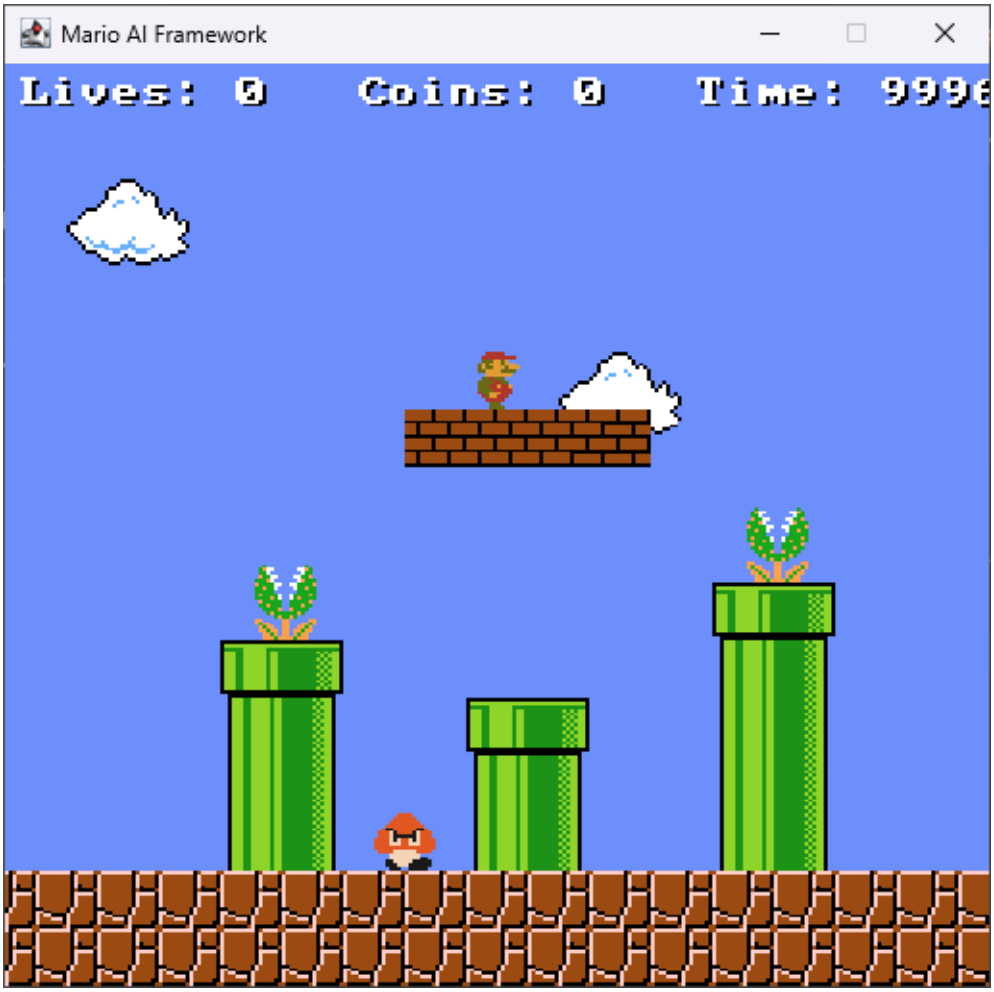


Figure 5.2: An example of a level segment from level 4 of the `original` level pack.

5.2.2 Mario physics

We want the grid path, which will guide our search, to match the way Mario would move through the level as precisely as possible. To be able to achieve that, we first need to analyse how Mario's movement works and which moves and transitions between moves are possible.

Speed and momentum

The first issue we need to deal with is that while Mario is running, he progressively speeds up (up to a certain point) and he carries this momentum when he jumps or falls. This means that Mario is able to jump way further if he first runs for the distance of 10 blocks than if he starts jumping from a standstill. We have manually tested the distances that Mario can jump given different amounts of blocks to gain momentum. Some of the results can be found in Table 5.1. When measuring the horizontal jump distance, we were always landing at the same height as the one where the jump began.

For the implementation of the grid search, we have decided to use the jump force that corresponds to running 2 blocks before jumping. The reasons are that

Blocks ran before jump	Jump distance (gap size)
1 block	7 blocks
2 blocks	9 blocks
4 blocks	10 blocks
14 blocks	11 blocks

Table 5.1: The distances that Mario can jump over while landing at the same vertical level. The left column shows how many blocks did Mario run before initiating the jump. The right column shows how many blocks can be jumped at maximum.

in most cases there are at least 2 blocks for Mario to gain speed before the jump and that this jump distance already means jumping so far that we do not see where we are landing (by quite a lot).

Jump manoeuvering

Jumping is even more complex because we might not only want to jump forward, but also upwards. And we might also want to change the direction of the jump.

As jumping up is concerned, the maximum height that Mario can jump is 4 blocks, meaning that Mario will get to a block that is 4 blocks higher than the block he was at when the jump started.

We will often want Mario to jump diagonally, therefore we manually measured how far can Mario jump while also landing higher than where he started. All of these measurements were already taken with the 2 tile start for gaining speed. The results are in Table 5.2.

How many blocks up do we jump	How far can we jump (gap size)
0 blocks	9 blocks
1 block	8 blocks
2 blocks	7 blocks
3 blocks	6 blocks
4 blocks	5 blocks

Table 5.2: A table showing how big of a gap can we jump over given that we also want to ascend. This measurement is done using the 2-tile start for gaining speed.

Figure 5.3 visualises these measurements by showing which tiles can Mario reach from his initial position, assuming the 2 tile start.

We have also tested how much can we manoeuvre midair. It is almost impossible to change the direction of the jump once it started because we immediately lose all momentum and do not really gain any in the opposite direction before we land. The only exception is jumping up on a block at the same horizontal position as where we start the jump. This move is very important in some cases and requires starting the jump in one direction and ending it in the other. An example can be seen in Figure 5.4. We took care in the grid search implementation to enable this move.

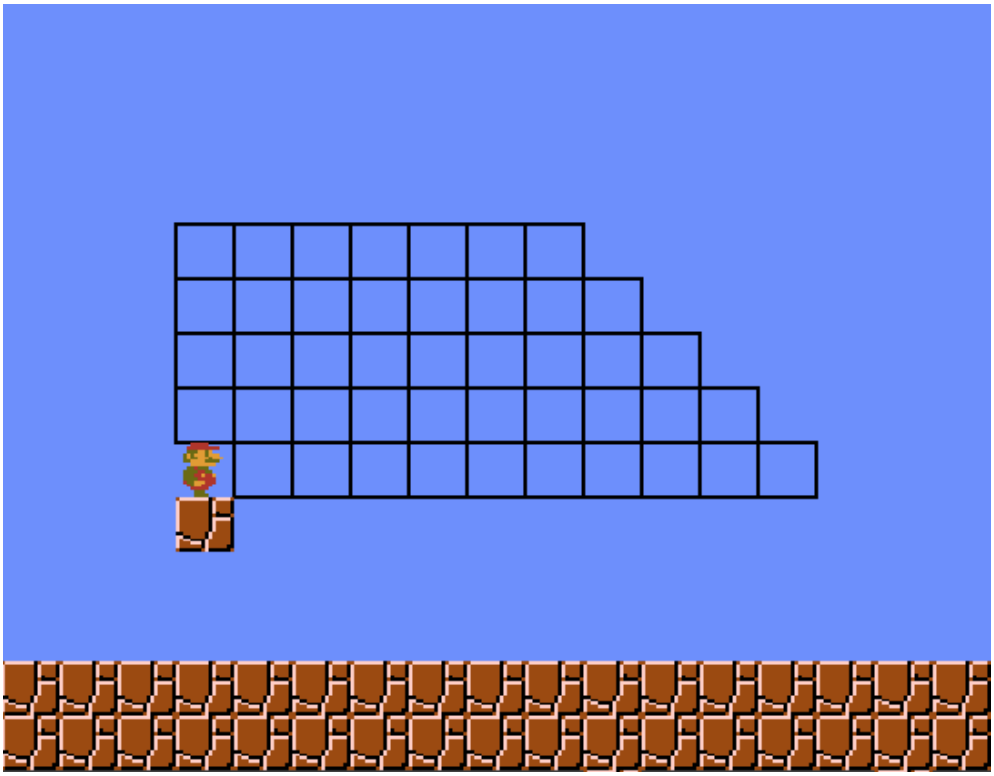


Figure 5.3: This figure visualises which positions can Mario reach by jumping from a 2-tile running start. The black squares represent tiles where Mario could land if there was a block underneath.

Free-falling

Besides jumping, we also looked at how we can control Mario when he is falling down. If Mario has no initial sideways energy, it takes a very long time to gather some, making it very hard to travel horizontally. If Mario starts falling with a decent amount of sideways energy, he can travel a few blocks before landing, if the fall is long enough. Attempting to change the sideways direction of falling makes Mario start falling straight down, but we will not manage to travel in an opposite direction before landing.

All of this taken into account, the free-falling behaviour is quite restrictive. In our implementation of the grid search, we have made the free-falling a bit more powerful to ensure that all levels can be solved.

5.2.3 Implementation

The implementation of the grid search is done using the A* algorithm. But because of the complexity of Mario's movement that we just described, we have decided to approach the state advancing of the A* search using a custom state diagram, which can also be thought of as a deterministic finite automaton. The high-level idea of the algorithm can be seen in Listing 5.1.

Listing 5.1: Grid search using the A* algorithm

```
function findGridPath() {
    while (opened.size > 0) {
```

```
            current = opened.remove(); // node from opened with the
                lowest cost
            if (finished(current)) {
                return getPathToNode(current);
            }
            for (action : getPossibleActions(current)) {
                  newState = advance(current, action);
                  heuristicValue = goal.X - newState.X;
                  newState.cost = newState.depth + heuristicValue;
                  if (visitedCosts[newState] != unknown) {
                      if (newState.cost >= visitedCosts[newState])
                        continue;
                  }
                  visitedCosts[newState] = newState.cost;
                  opened.add(newState);
            }
        }
        return getPathToNode(furthestNode); // path to goal not found
    }
```

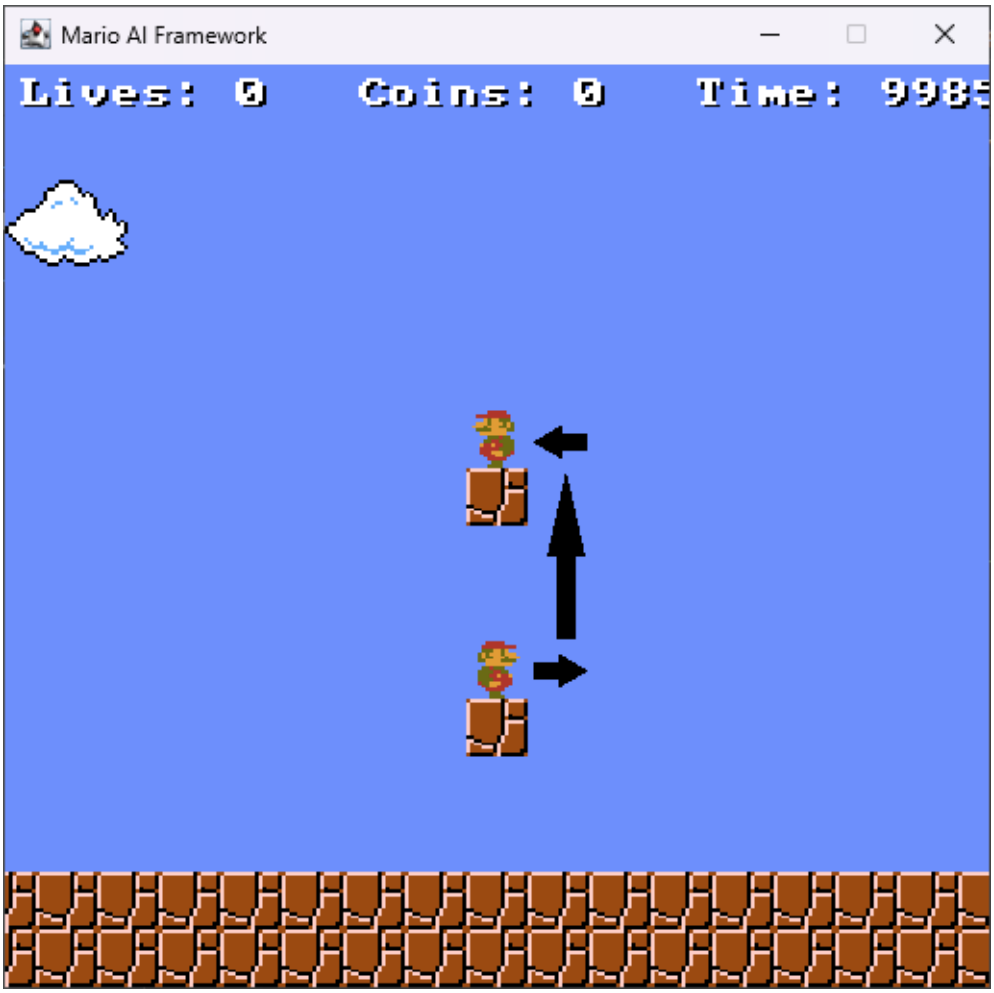


Figure 5.4: Visualisation of the path Mario will take to jump up higher while landing on the same horizontal position.

We needed to model the complex Mario's movement possibilities concerning his momentum and air manoeuvrability. For that, we created a state diagram. In each node of the search (each tile), we would know which movement state Mario is in and based on that we could determine his possible actions and advance the state. Besides the movement state, we also tracked a few other variables, such as jump height or horizontal distance travelled, to help us accurately simulate the movement that we described in Subsection 5.2.2.

The state diagram is unfortunately too complex to be drawn reasonably, but the implementation of it can be found in the source code repository[1] in the file `GridSearch.java`. It is implemented using the `switch` expressions, which actually makes the source code of it quite well-readable.

In the rest of this section, we will go over some of the implementation details that we needed to handle.

Jump up on the same horizontal position

As we found out during the analysis of Mario's jump manoeuvrability 5.2.2 and as is illustrated in Figure 5.4, we needed to allow this special kind of jump in our search.

Since we did not allow changing the direction of a jump midair, we had a special state called `WALKED_OFF_AN_EDGE` to tell us that Mario left solid ground. In this state, we did not yet define the direction of the jump, but rather left a possibility to continue the movement straight up. During this movement up, the jump direction still did not need to be decided. Finally, when the search would go to move left or right, we would define the jump direction and only continue this way.

Now, by first going into the `WALKED_OFF_AN_EDGE` state when leaving solid ground, then jumping straight up and landing on the same horizontal position, but a few blocks higher, we can accurately simulate the desired move.

Running over a 1 block wide pit

When we were done implementing the search, we noticed it failing in a situation shown in Figure 5.5. Here, Mario needs to get over a 1 block pit, but can not jump up because of the low ceiling. Even though one might consider this situation unsolvable, if Mario gathers enough speed, he can use his momentum to run over the pit.

Once again, this situation was solved using the `WALKED_OFF_AN_EDGE` state, from which we allowed the search to move one block to the side to reach solid ground again.

Jump boost

As shown in Table 5.1, Mario can jump further if he gathers speed over longer periods of running forward before the jump. And although we have decided to implement the jump strength in our search as if Mario would always have 2 tiles to gather momentum, we wanted a way to increase the jump strength on demand. To enable this, the `GridSearch` class can be initialized with a `horizontalJumpBoost`

[1] `https://gitlab.com/gamedev-cuni-cz/theses/super-mario-astar`

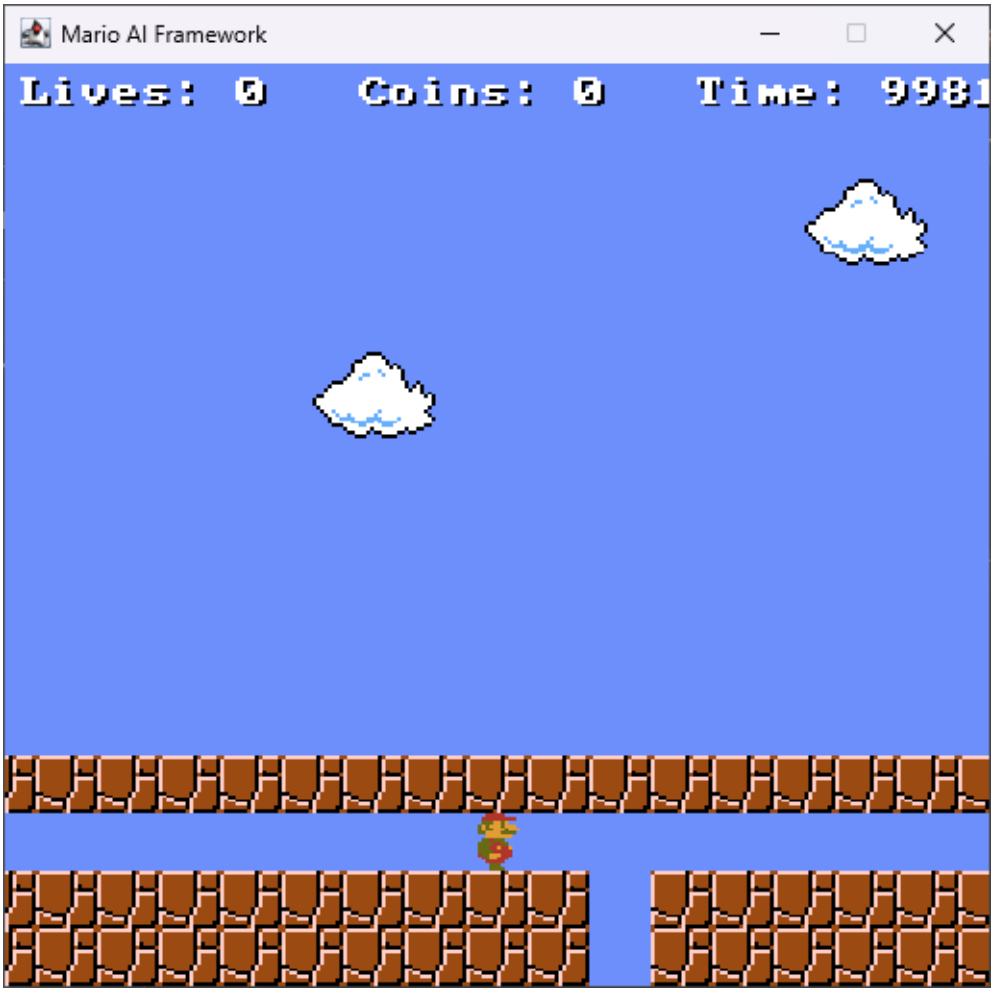


Figure 5.5: A situation that might seem unsolvable at first, but if Mario gathers enough momentum, he can run over the pit.

parameter that specifies how much further we want Mario to be able to jump compared to the standard jump distance we use.

This can be used to simulate giving Mario more blocks to gather speed before each jump. Even though this option is present in our implementation, we never had to use it to solve a level, since no sane level would force a player to jump so far (because from a certain point, the landing spot is far outside the visible screen).

Semisolid Platform

Most blocks in Super Mario Bros. are either solid, meaning that they block movement from all sides, or empty, and can be walked in from any direction. But there is a special type of blocks that we need to deal with – semisolid platforms. These are blocks that block movement when approached from above (therefore function as solid blocks), but can be passed through from below. The usual scenario is that Mario jumps up through the semisolid platform from below to get on top. These semisolid platforms can also be walked in from a side, given they are placed on top of another solid block.

When implementing the grid search, we employed special checks when gather-

ing available moves in a state to ensure that we handle semisolid blocks correctly.

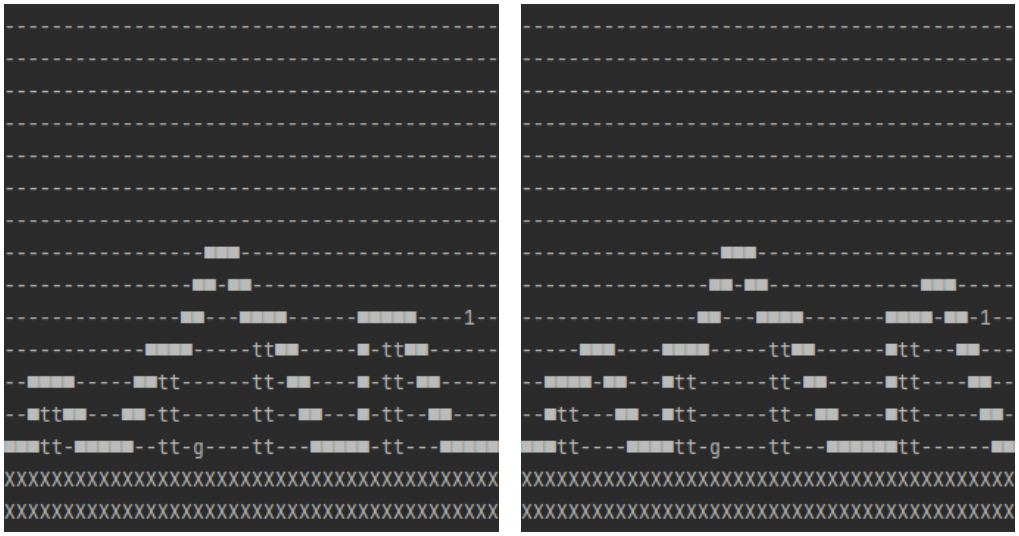
Enemies

In the grid search, we completely ignore enemies. Their positions are not even passed into the search. As the grid path is meant to guide the agent's search in a more high-level way, it is not needed to handle enemy dodging here at all, as the low-level search can handle that really well.

Heuristic function weight

As the grid search is implemented using the A* algorithm, it uses a heuristic function. The heuristic function is implemented as the distance from the current tile to the goal one. Let us look at the difference between using a heuristic function of weight 1.0 (standard value) and 2.0 (weighted A*). We tested the grid search on the first level from the `original` level pack, which consists of the original levels present in the game.

With a heuristic function weight of 1.0, the grid search visited 15 339 nodes to reach the goal. With a heuristic function weight of 2.0, the search only needed to expand 440 nodes. This might lead us to the conclusion that greater weight is better, but let us first compare the paths found. The first interesting segments are compared in Figure 5.6 and the second interesting segments are compared in Figures 5.7 and 5.8.



(a) The grid path found with heuristic function weight of 1.0.

(b) The grid path found with heuristic function weight of 2.0.

Figure 5.6: A comparison of the grid paths found with different heuristic function weights settings in the grid search.

We can see that with a weight of 1.0, the path tends to nicely approach jumps up on a pipe or a pyramid, but unnecessarily walks down from them rather than jumping forward. With a weight of 2.0, the path sometimes hugs a pipe before jumping onto it, but it has better jumps forward when leaving higher ground. For our agents, we ended up using the value of 1.0, since we do not mind taking

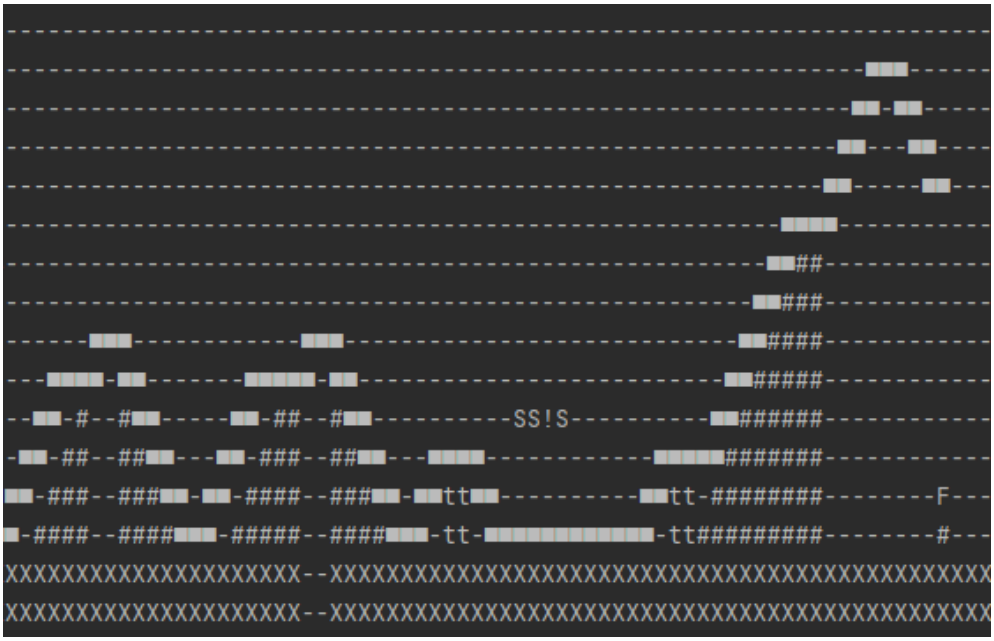


Figure 5.7: The grid path found for the final segment of the first `Mario` level by a grid search with heuristic function weight of 1.0.

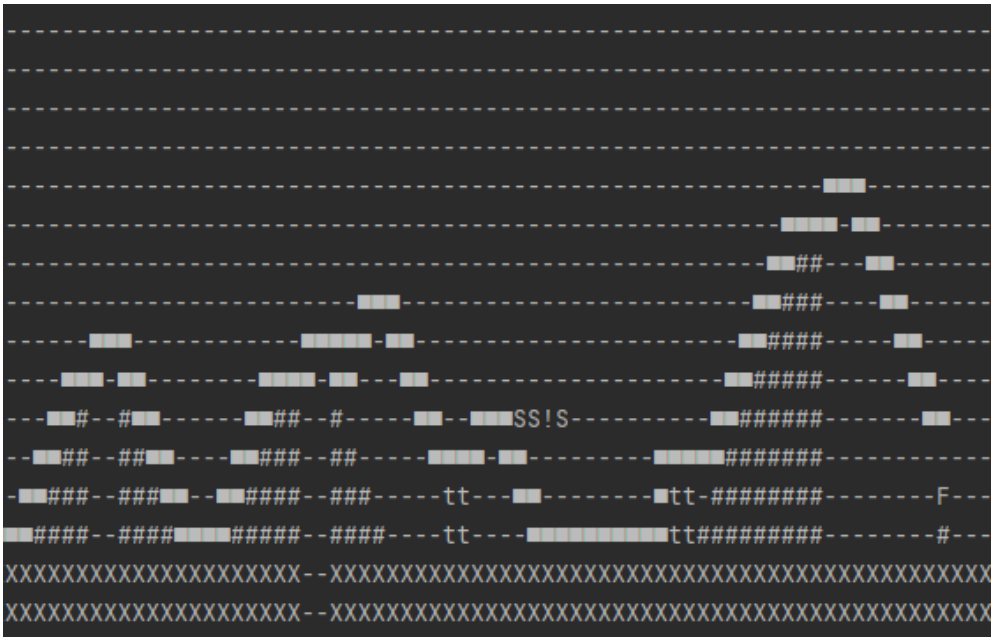


Figure 5.8: The grid path found for the final segment of the first `Mario` level by a grid search with heuristic function weight of 2.0.

a bit longer once to find the grid path when initializing, and since the differences in the paths are not significant.

Comparison to Mario physics

The correctness of the grid search is tested in Section 6.3. But we also want to look at how accurately we managed to model Mario's physics. We dare to say that we managed quite well. The most data-supported evidence is that an agent

utilizing the grid path was able to solve all levels in our test pack, and did so in very good times as well (for details, see 6.4).

Apart from that, we can say that from our observations of the paths produced by the algorithm, they appear exactly as we hoped to and as we imagined an agent or a player would navigate the levels. As far as we know, the only mismatch between the grid paths and actual Mario physical capabilities is free-falling. Here, we ended up allowing the grid search to be a bit stronger – letting it change directions more easily, to ensure that it always finds a path to the goal. We are not aware of this ever causing problems in the levels we experimented with.

5.3 MFF A* Grid agent

With the grid search implemented, we can now utilize the grid path that we made available to implement a new agent. This agent is based on the previous state-of-the-art agent called `MFF A*`, which is described in detail in our previous work [1]. It is an A*-based agent that searches among game states, starting from the current tick and trying to reach a state where the level is finished. It uses Mario's actions as state transitions.

The core of the agent implementation is the same as for the `MFF A*` agent. The agent is called every tick and required to return an action that Mario will take. The game runs at 30 frames per second, thus the search gets approximately 33 milliseconds to produce an answer. Every time, a new search is initialized, starting at current Mario's position. After time runs out, we look at the farthest position towards the right that Mario has reached in any of the visited nodes of the current search, compare it with the farthest one we had found so far, and save the actions sequence that leads to the further one. Then we return the action leading towards the farthest found state.

A trick that we employ is to not apply an action only once when advancing to a new state but to apply it 3 times. This leads to a significant reduction of the search space, and it allows Mario to change the action he is performing 10 times per second, which is still at a superhuman level.

The main change we made to the `MFF A*` agent to get our new `MFF A* Grid` agent was to change the cost function of the search nodes so that it utilizes the grid path that we can now obtain using the implemented grid search.

5.3.1 Cost function

First, we will talk about how the cost function was implemented in the `MFF A*` agent, and then we will present the implementation for the `MFF A* Grid` agent.

MFF A* cost function

Listing 5.2 shows how the `MFF A*` agent from our previous work calculates the cost of a node. We use `timeToFinish` as a heuristic function, which is calculated as the horizontal distance from Mario's current position to the goal position divided by Mario's maximal horizontal speed. As a result, we obtain the time in game ticks that Mario would need to reach the goal if he ran at maximal speed in a

straight line. This heuristic function is admissible, as it represents the lowest possible time to reach the goal.

Listing 5.2: The node cost calculation of the `MFF A*` agent.

```
float calculateCost(node) {
    timeToFinish = (exitTileX - node.state.marioX)
        / maxMarioSpeedX;
    return node.depth + timeToFinish * 1.1;
}
```

To obtain the node cost, our previous work multiplies the heuristic function by 1.1 to make the search prefer states that are closer to the goal (this means that we are using weighted A*), and sums it with the node's depth, which corresponds to the number of ticks we took to reach the node from the origin of the search.

As the action selected for the transition to the next state is applied 3 times, but this is not compensated in the cost function, the weight of the node cost is effectively $\frac{1}{3}$, which makes the heuristic very strong.

Now that we know how the cost function was calculated for the `MFF A*` agent, let us now utilize the grid path to improve the agent and obtain the `MFF A* Grid` agent.

MFF A* Grid cost function

We will use the grid path as a corridor leading to the goal state, which we will try to stay close to. The cost function calculation is shown in Listing 5.3.

Listing 5.3: The node cost calculation of the `MFF A* Grid` agent.

```
float calculateCost(node) {
    timeToFinish = (exitTileX - node.state.marioX)
        / maxMarioSpeedX;
    pathDistance = calculateDistance(node);
    if (pathDistance <= distanceTolerance)
        distanceFromGridPathCost = 0;
    else
        distanceFromGridPathCost =
            (pathDistance - distanceTolerance)
            * distanceMultiplicativePenalty
            + distanceAdditivePenalty;
    return nodeDepthWeight * node.depth
            + timeToFinishWeight * timeToFinish
            + distanceFromGridPathCost;
}
```

The cost function of `MFF A* Grid` consists of three parts: node depth, time to finish heuristic and distance from grid path heuristic. The first two parts are calculated in the same way as for the `MFF A*` agent. The distance from grid path heuristic works as a penalty, which is applied when the agent is too far away from the path.

First, `pathDistance` is calculated as the distance from the nearest block that lies on the grid path. Then, there is a `distanceTolerance` parameter that we

use to simulate different widths of the path corridor. If the parameter is set to 0, the agent needs to be present directly on the path to not get penalized. If we set the parameter to 1, the path is effectively 3 tiles wide, therefore the agent can be 1 tile away from the original path and still will not get penalized. It works the same for bigger values of the parameter. Finally, if the agent is too far away from the path, even with the tolerance taken into account, it gets penalized (otherwise, the value of `distanceFromGridPathCost` is set to 0).

We have two types of penalties. First, a `distanceAdditivePenalty` is applied whenever the agent is too far away from the path. This penalty is applied once no matter how far away the agent is. Second, there is a `distanceMultiplicativePenalty`. This penalty is multiplied by the `pathDistance` reduced by `distanceTolerance`, so it is applied per every tile that the agent is beyond the corridor created by the `distanceTolerance` parameter.

Note that path distance is calculated as the distance of the agent to the original single-tile-wide grid path, that is why we subtract `distanceTolerance` from it to obtain the actual distance from the simulated path.

Parameter optimization

In the `MFF A* Grid` cost function description, we mentioned the three parameters that specify the `distanceFromGridPathCost` calculation – `distanceTolerance`, `distanceAdditivePenalty` and `distanceMultiplicativePenalty`. There are two more parameters in the cost function – `nodeDepthWeight` and `timeToFinishWeight`. In total, this is a set of 5 parameters that influence the behaviour of the agent and the strength of the heuristic functions.

We decided to run an extensive parameter search to optimize these 5 parameters. As with the parameter search for `MFF A*`, we sampled various parameter sets and tested them on the 1015 levels we have available (100 per level generator for the Mario AI framework and the 15 original Super Mario Bros. levels). We tested all possible combinations of the following parameter values:

- `nodeDepthWeight`: $\{ 0, 0.5, 1, 1.5, 2, 3 \}$
- `timeToFinishWeight`: $\{ 0, 0.5, 1, 1.5, 2 \}$
- `distanceTolerance`: $\{ 0, 1, 2, 3, 4, 5, 7, 10, 15, 20 \}$
- `distanceAdditivePenalty`: $\{ 0, 1, 2, 3, 5, 10, 20, 50 \}$
- `distanceMultiplicativePenalty`: $\{ 0, 1, 2, 3, 5, 7, 10, 20, 50 \}$

This meant a total of 21 600 parameter sets. For each of them, a job was sent to the MetaCentrum cluster. The results are evaluated and discussed in Section 6.4.

5.4 MFF A* Waypoints agent

Even though the new `MFF A* Grid` agent manages to tackle all of the standard levels really well (as we will see in Chapter 6), it is not able to solve large mazes as its heuristic function pushes it to always go to the right. This is completely

usable for standard levels, where the path goes mostly straight from left to right – and that is the case for both the original 15 Super Mario Bros. levels, as well as for all of the 10 level generators we use to test our agents on.

But to enable an agent to solve large maze-like environments, we need to adapt it further. This section will present a proof of concept that such agent could be created by tweaking the `MFF A* Grid` agent. We have created a showcase level specifically to test this proof of concept agent – a level that does not flow straightly from left to right throughout the playthrough, but it requires long periods of going to the left. It is also essentially a huge maze, therefore the agents need to know which direction to go, or else they might get stuck somewhere inside. The level was already presented in Subsection 4.1.2, but we show how it looks once more in Figure 5.9.

Figure 5.9: The showcase level created to test how agents can handle maze-like environments.

Only a part of the level can be seen, but the rest of the level looks in the same manner. There are hallways of alternating directions, which are more than 100 tiles long. The agent starts at the bottom left corner and needs to get to the top right corner to escape the maze and reach the finish. To do so, it must alternate running to the right and to the left while reaching one floor higher every time it can.

5.4.1 Implementation

To solve this level, we needed to tweak the way that the `MFF A* Grid` agent works to allow it to navigate through the maze in the correct order. The main trick is to introduce waypoints, which is why the new agent is called `MFF A* Waypoints`. The pseudocode of the agent's search loop can be seen in Listing 5.4.

Listing 5.4: The main loop of the agent's search.

```
while (opened.size > 0 && remainingTime > 0) {
        current = opened.remove(); // get current node
        if (isGoalState(current.state)) {
            return (getPathToNode(current));
        }
        if (getDistance(current, nextWaypoint) < bestDistance) {
            furthestNode = current;
            bestDistance = getDistance(current, nextWaypoint);
        }
        if (waypointReached(current,
            WAYPOINT_HORIZONTAL_DISTANCE_TOLERANCE,
            WAYPOINT_VERTICAL_DISTANCE_TOLERANCE)) {
            deleteGridPathToWaypoint(nextWaypoint);
            nextWaypoint = getNextWaypoint();
            furthestNode = current;
            bestDistance = getDistance(current, nextWaypoint);
        }
        for (action : getPossibleActions(current)) {
            newState = current.clone();
            newState.advance(action);
            if (newState.isMarioAlive() == false)
                continue;
            float newStateCost = calculateCost(newState);
            if (getnewStateOldCost() != unknown && newStateCost >=
                getnewStateOldCost())
                continue;
            saveNewCost(newStateCost);
            opened.add(newState);
        }
    }
    furthestSafeState = getFurthestSafeState(furthestNode);
    return getPathToNode(furthestSafeState);
```

The waypoints represent subgoals that help lead the agent in the correct direction. A waypoint is simply a position in the level. The waypoints are sampled from the grid path that we have available, starting from the initial Mario's position, creating a waypoint every few blocks of the grid path until we reach the finish. The waypoints are always sampled to be on the ground so that the agent does not need to collect them mid-air. We can see an example of where the waypoints might be in Figure 5.10. Here, we sample a waypoint every 8 blocks. Mario is currently pursuing the waypoint to his right and after he reaches it, the subgoal will shift to navigate him to the next waypoint. The pseudocode of the waypoint sampling can be seen in Listing 5.5.

Listing 5.5: Pseudocode of waypoint sampling.

```
void initializeWaypoints(gridPath) {
    int waypointsSpacing = WAYPOINT_DENSITY - 1;
    for (node : gridPath) {
        waypointsSpacing++;
        if (waypointsSpacing < WAYPOINT_DENSITY)
            continue;
        if (isOnGround(node)) {
            waypoints.add(new Waypoint(node.X, node.Y))
            waypointsSpacing = 0;
        }
    }
    if (waypoint.contains(lastGridPathNode) == false) {
        waypoints.add(lastGridPathNode); // include goal tile
    }
}
```

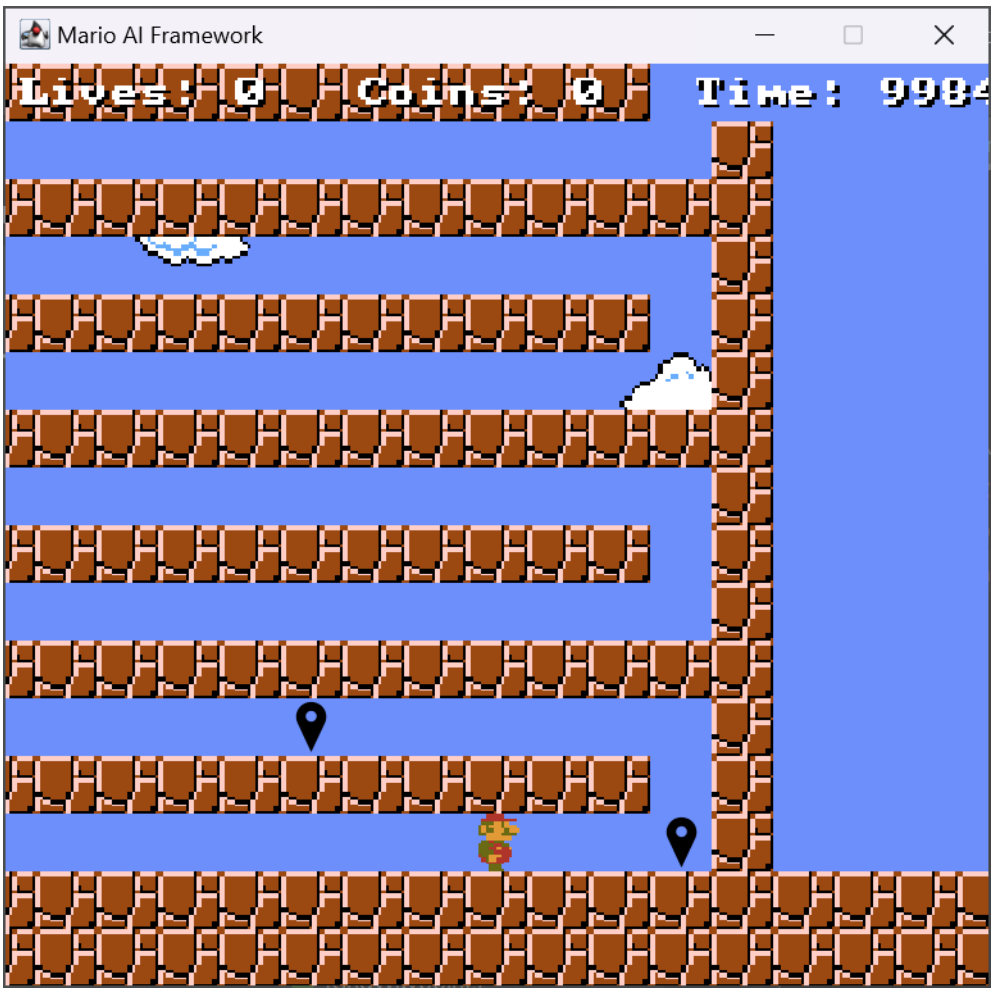


Figure 5.10: A visualisation of how waypoints may be sampled from a grid path.

The waypoint density of 8 is what we actually used in our implementation. The value was manually selected. The reason for sampling the waypoints this often is to avoid problems in a situation where the next waypoint would be

directly above the agent, but on a different floor. If the waypoints are close enough to each other, the search can handle such a situation.

Heuristic changes

Because the agent still utilizes the grid path distance heuristic when navigating to a waypoint, we use another trick to make this more efficient. When a waypoint is reached, the grid path from the previous waypoint up to the currently reached one is deleted. This helps to keep the agent on the right path, as the tiles in the opposite direction will have even higher penalization because the grid path is no longer present there.

Another thing that we needed to change was the calculation of the `timeToFinish` component of the cost function. Instead of calculating the time necessary to reach the goal of the level, we return the time necessary to reach the next waypoint. By this, we eliminate the agent's urge to always run right.

Waypoint tolerance

Because forcing the agent to reach an exact position could be problematic, there is a tolerance of how far from the waypoint can an agent be for it to count as reached. We set these tolerances to 16 pixels (width of 1 tile) horizontally and 4 pixels vertically. The vertical tolerance is lower because we do not want to accidentally reach a waypoint on a different floor. At the same time, the horizontal tolerance is small enough to not allow picking up a waypoint over a thin wall.

Planning

The last change is to not start the search from the current Mario's position, but from the farthest position reached last search (towards the next waypoint). This allows us to preplan our path more ahead and gives us more time to handle branching. We take care to not necessarily take the furthest node, but the furthest one that is safe (where Mario is on the ground) to ensure that the new search can start without any trouble. A similar idea was used in an agent called `MFF A* - dynamic planning`, which is described in our previous work [1].

5.4.2 Summany

This agent can handle the showcase level described above flawlessly, as the only agent we have available, thus proving that solving even non-standard levels is possible using our approaches. However, as this agent is only a proof of concept, it was not tweaked to perfection, therefore it may fail on some of the standard levels.

One upgrade that we think might help would be to reset the search algorithm once a new waypoint is reached, because reaching a new waypoint changes the costs of nodes and we may potentially have quite a lot of nodes opened that have their cost calculated using an old waypoint. Another step might be to experiment with waypoint density, waypoint distance tolerances, and all the parameters that we have inherited from `MFF A* Grid`.

6. Evaluation

In this chapter, we will evaluate all of our work. First, we will present the testing environment setup in Section 6.1. We will then look into the parameter search for the `MFF A*` agent in Section 6.2. Following that, we will test the correctness of the grid search that we implemented in Section 6.3.

The `MFF A* Grid` agent archetype created using the grid search is evaluated and its best parameters are chosen in Section 6.4. After that, we will try to find the most universal Super Mario Bros. agent in Section 6.6.

The experiment data that this chapter is based on can be found at `https://gitlab.com/gamedev-cuni-cz/theses/super-mario-astar`.

6.1 Experiment setup

This chapter will first describe the testing environment, then it will present the level packs that we will be using to test agents' performance. It will also address the fact that, as we found out, some of the levels in our previous tests [3] were not solvable.

6.1.1 Testing environment

As described in Chapter 3, we used the computational resources of the MetaCentrum organisation to run our experiments. We needed to ensure that all of the experiments were run on the same hardware to allow for a fair comparison among individual parameter sets and also with the previous state-of-the-art agent.

We achieved this by selecting the computational nodes that we wanted for our experiments and requesting them using the options of the PBS (portable batch system) through which we submitted the jobs. We evaluated all of our experiments on `gita` nodes with a SPEC CPU® 2017 [10] rating of 5.1 points.

6.1.2 Level packs

This section will present the levels that we use to evaluate the agents. We use the 15 levels that are originally part of the Super Mario Bros. game, and then 10 different levels generators, from each of which we use 100 generated levels. That means we are using 1015 test levels in total. We refer to each of the level sets that comes from a single source as a level pack.

Let us now talk about the individual level packs.

`Mario/original`

These are the 15 levels originally present in the Super Mario Bros. game. We refer to them as the `original` levels or as the `Mario` levels.

The `Mario` levels are simple in the sense that they do not feature any branching or maze-like parts. There are two types of them – let us call them `ground` and `aerial`. The `ground` levels have most of the floor solid, occasionally featuring a pit that needs to be jumped over, but they feature quite a lot of different

enemies. The `aerial` levels are made up of individual platforms with no solid ground below, so they are more jump intensive.

`krys`

The `krys` level generator works by stitching together manually created level segments. The pool of the segments is quite large and it features a lot of interesting problems, including small maze-like parts. The level pack is also quite hard for a human player, as it features intricate parts with little manoeuvring space and many enemies.

Because of this, we will pay special attention to agents' performance on this level pack.

`ge`

This level pack is very easy as the levels are mostly narrow with little enemies and only a few jumps.

`hopper`

Another easy level pack – the levels mostly consist of platforms and sometimes a big group of enemies appears. But beware, because this level pack can get unexpectedly tricky, where it rarely happens that the path branches and only one of the paths leads to the finish, while the other is a dead end.

`notch, notchParam, notchParamRand`

Levels with a single mostly straight path, featuring easy jumps and few enemies.

`ore`

These levels often contain two parallel paths above each other, mostly made out of platforms. It seldom happens that only one of the paths is correct.

`patternCount, patternOccur, patternWeightCount`

For most of the levels, there is a huge amount of small gaps between blocks. At the end of the levels, it sometimes happens that extremely long jumps are necessary.

6.1.3 Unsolvable levels

From our previous work [3] we knew that not all of the levels present in the framework are solvable. The levels were generated using various level generators, and we noticed that some of them tend to generate levels that can not be completed. We have gone through all of the levels and fixed those that needed it, trying to change the levels as little as possible.

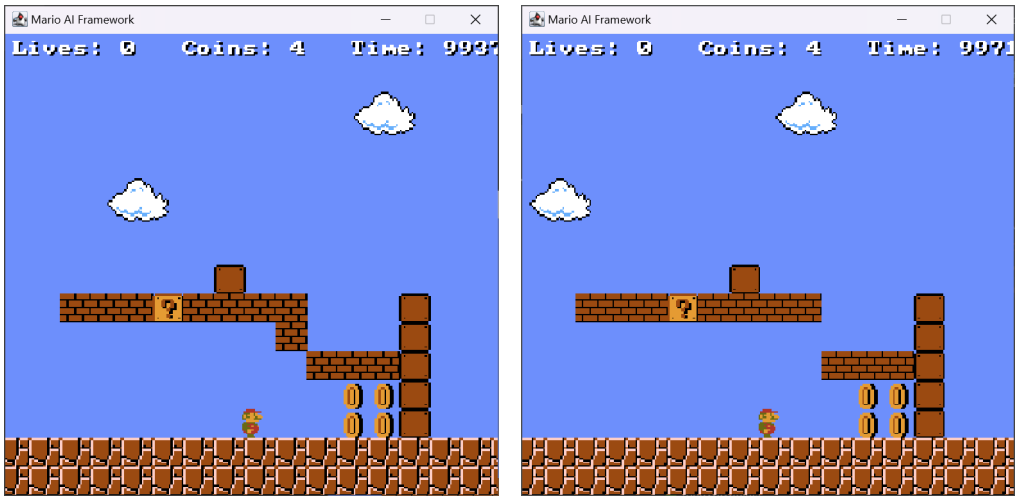
We have fixed 1 `krys` level, 21 `patternCount` levels, 26 `patternOccur` levels and 17 `patternWeightCount` levels. All of the agents present in this work will be tested on these fixed levels, including the best agent from our previous work [3],

which will be used for comparison (and whose parameters were tuned as a part of this work, see Section 5.1).

Levels requiring brick breaking

Some levels from the `krys` level pack contained a segment that required the agent to break a destroyable brick in order to proceed. Since Mario is only able to break bricks after collecting a power-up, and collecting power-ups is something we do not interest ourselves with at all, we have decided to remove such obstacles from the test levels completely.

We have edited the 6 `krys` levels (out of 100 generated) that contained the mentioned breakable brick segment, always doing so in the same manner by removing one of the bricks, not editing the levels any further. Figure 6.1 shows how this edit looks in one of the levels.



(a) Level segment before edit, path further is not possible without block breaking.

(b) Proceeding further is now possible without block breaking.

Figure 6.1: The level editing we did for the `krys` levels to remove the need to break blocks.

6.2 MFF A* agent parameter search

As described in Section 5.1, we wanted to find the best parameters for the `MFF A*` agent. The two parameters that we are optimizing are node depth weight (NDW) and time to finish weight (TTFW). We are testing values from 0.2 to 4.0 with a step of 0.2 for NDW and values from 0.2 to 2.0 with a step of 0.2 for TTFW. As already mentioned, we use each selected action three times, which means that the NDW value is effectively divided by 3. We decided not to change the implementation in order to be consistent with our previous works.

We tested 200 parameter sets in total (all possible combinations of tested weight values). For each of the parameter sets, we ran the agent on the 1015 levels that we have available – 100 per level generator and 15 Super Mario Bros. ones. The following metrics were collected for each level:

- `win/fail` – whether the agent completed the level.
- `travel` % – percentage of distance travelled from start to goal.
- `run time` – the total time of the test, including agent searches and game updates.
- `game ticks` – the number of ticks the agent needed to win/fail the level.
- `planning time` – total time of the agent searches.
- `total planning` – number of agents searches executed.
- `nodes evaluated` – total amount of nodes the agent explored.
- `most backtracked nodes` – the largest amount of nodes evaluated before a new furthest state towards the goal was found.

The correlation matrix of the parameters and the metrics across all levels and all parameter sets can be seen in Figure 6.2. We will now look at the results for the various metrics collected.

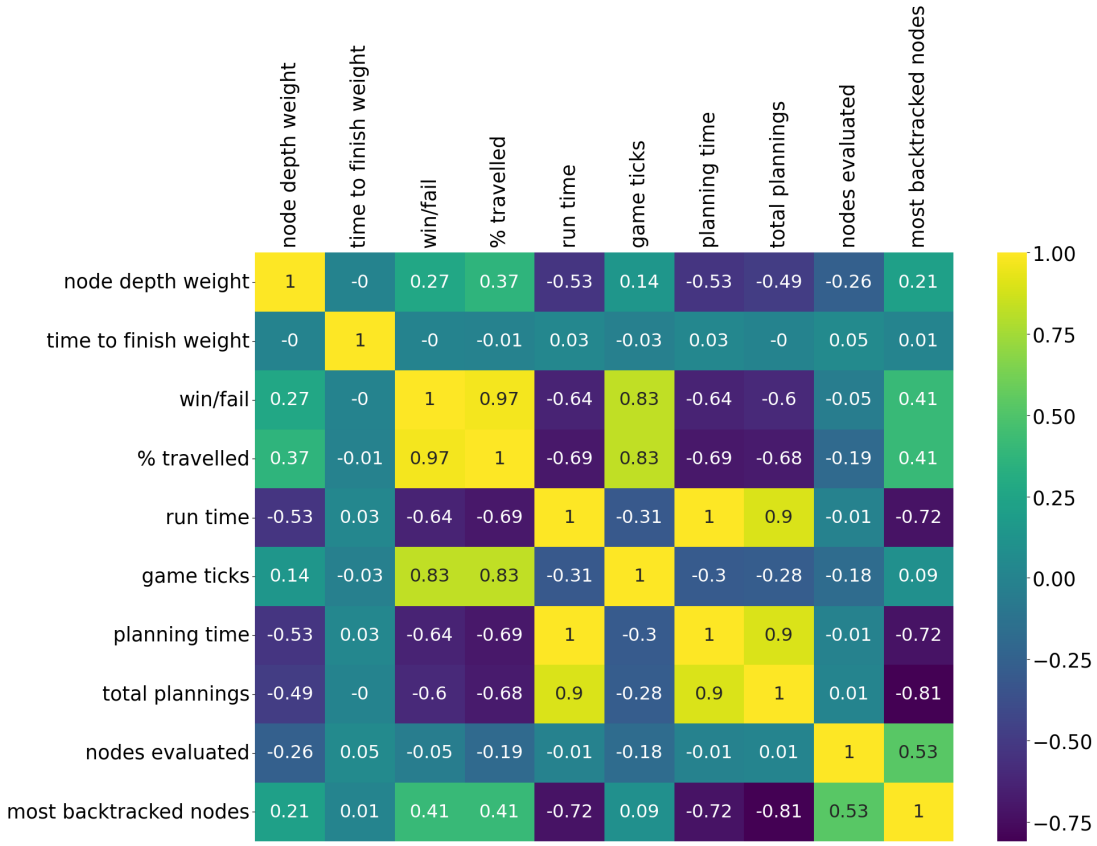


Figure 6.2: Correlation matrix of the `MFF A*` agent's parameters and metrics across all levels and all parameter sets.

6.2.1 Win rate

The lowest win rate (on all 1015 levels in total) was 97.34 and the highest was 99.21 (rounded to 2 decimal places). Surprisingly, all of the configurations did quite well and the differences in win rate are really low. Please note that these win rates can not be directly compared with our previous results, as we have fixed some levels to be solvable.

As for the dependence of win rate on the parameters, the correlation of win/fail and node depth weight is 0.27 and the correlation of win/fail and time to finish weight is −0.00 (rounded), therefore the influence of both of the parameters can be seen as negligible. This comes as a big surprise, but really, both for the highest and the lowest win rates, there are significantly different weight ratios right next to each other in the results.

A complete overview of the win rate for different parameter sets can be seen in Figure 6.3.

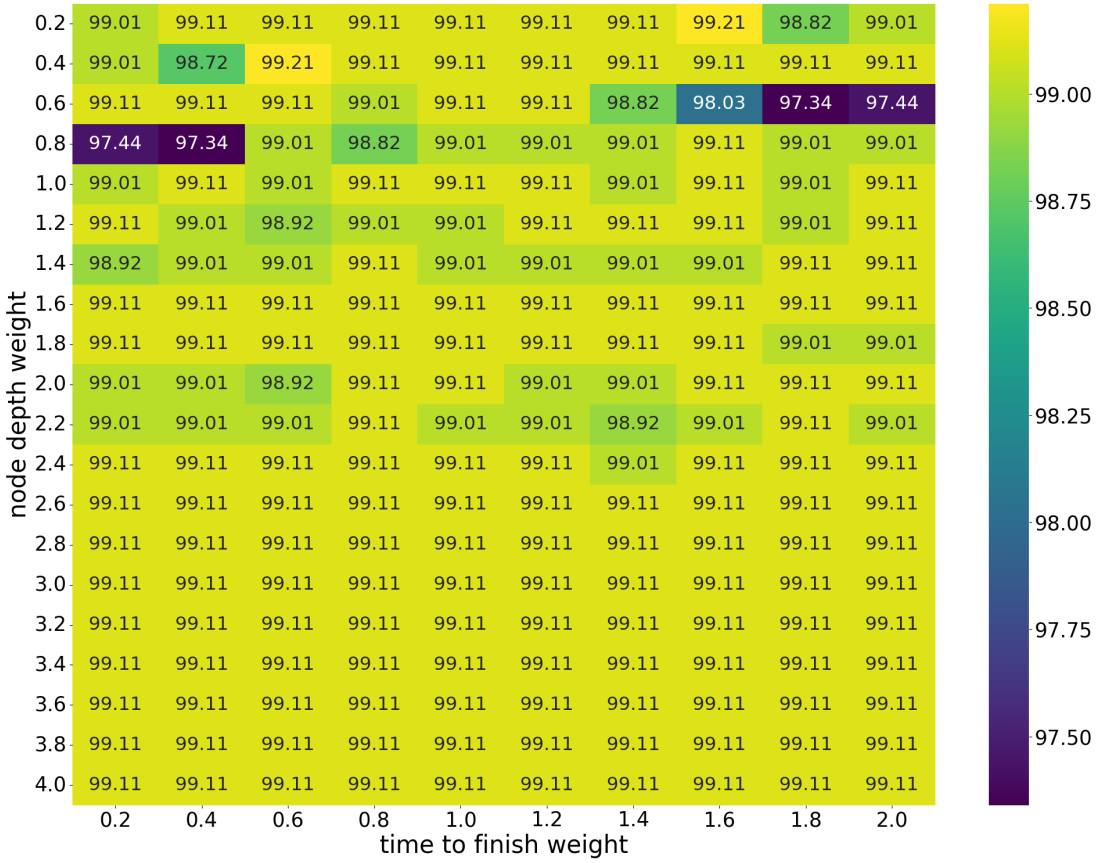


Figure 6.3: A graph showing the total win rate for all of the parameter sets tested.

6.2.2 Nodes evaluated

The best result for nodes evaluated (average over all levels) was 49 689 nodes, and the worst result was 71 924. This difference can be significant, as the number of nodes that need to be evaluated influences the run time of the agents.

The correlation of nodes evaluated and node depth weight is −0.26 and the correlation of nodes evaluated and time to finish weight is 0.05 (rounded). Once

again, we do not see a strong correlation. From looking at the data more closely, we have observed that the 15 parameter sets with the lowest amounts of nodes evaluated all have NDW higher than TTFW, most of them more than twice as big.

This goes against our idea of higher TTFW meaning fewer nodes evaluated, thanks to the heuristic function being more aggressive, but it is important to note that the difference between the best and the worst evaluated nodes result is not that significant, as we will see agents with much lower and much higher average evaluated nodes win all of our test levels.

6.2.3 Other metrics

Let us now look at the remaining metrics and their correlation to the parameters.

If we look at the correlation matrix 6.2, the first thing to notice is that the TTFW parameter was completely insignificant in influencing the metrics. The reason might be that as the NDW parameter is effectively one-third of its value, the TTFW parameter is always somewhat significant, as its lowest value is 0.2. It seems that even if the heuristic function is quite small, it is still enough for the agent to be successful, and a significant increase in its strength does not necessarily mean that the agent will be more performant.

Looking at the correlation of the NDW parameter with the metrics, we can see that increasing the parameter has a non-negligible effect on decreasing run time, planning time and total plannings. Checking with the data, the first 11 parameter sets when filtering for the lowest run time really do have NDW significantly greater than TTFW.

The last metric we want to comment on is game ticks – the range of values for game ticks in our results is between 370 and 375. This means that none of the agents was able to find significantly faster paths through the levels.

6.2.4 Best agent

Out of the 200 different parameter sets that we have, it would be nice to know which of the agents can be considered the best one. To pick such an agent, which will later be used in a comparison with the other ones, we have decided to pick an agent with the lowest run time, given it also has a very good win rate.

Given these requirements, we choose the second-best parameter set when it comes to both run time and win rate. We could have chosen the one with the best run time and still the second-best win rate (the win rates are equal), but we liked the chosen agent's parameters more when it comes to their interpretation.

Therefore, the agent we will be using for the representation of the `MFF A*` agent archetype is the one with NDW of 4.0 and TTFW of 1.2. As mentioned many times, because of the implementation, the NDW is effectively one-third of its input value, so if we adjust the parameters we can think of the agent as if it had NDW of $1.\overline{3}$ and TTFW of 1.2. We like this choice because it is close to being a standard A* (no weighted heuristic).

6.3 Grid search correctness

With grid search 5.2 implemented, we needed to test its correctness – that is, if it can solve levels reliably. For that, we ran the grid search on 100 levels per each of the 10 level generators we have available and also on the 15 original Super Mario Bros. levels. The grid search was able to solve all of these levels.

Moreover, we wanted to explicitly test some of the Mario moves that we wanted the grid search to be able to perform. For that, we manually created a few custom levels. They are available in the repository of this project[1]. The grid search solved all of these levels too.

Since we did not find any level that the grid search could not solve according to our expectations, we consider it to be correct.

6.4 MFF A* Grid agent

In this section, we are going to talk about the results of the parameter search that we ran for the new `MFF A* Grid` agent.

The agent was tested on the same 1015 levels as the `MFF A*` agent in the previous section, but the number of parameters was higher. As we know from the implementation of `MFF A* Grid` 5.3, apart from the node depth weight (NDW) and time to finish weight (TTFW) parameters, which it inherited from `MFF A*`, it has three more parameters that influence the grid path heuristic – distance from path tolerance (DPFT, referred to as `distanceTolerance` in the previous text), distance from path additive penalty (DFPAP, referred to as `distanceAdditivePenalty` in the previous text) and distance from path multiplicative penalty (DFPMP, referred to as `distanceMultiplicativePenalty` in previous text).

We have tested all possible combinations of the following parameter values:

- `nodeDepthWeight`: $\{ 0, 0.5, 1, 1.5, 2, 3 \}$
- `timeToFinishWeight`: $\{ 0, 0.5, 1, 1.5, 2 \}$
- `distanceTolerance`: $\{ 0, 1, 2, 3, 4, 5, 7, 10, 15, 20 \}$
- `distanceAdditivePenalty`: $\{ 0, 1, 2, 3, 5, 10, 20, 50 \}$
- `distanceMultiplicativePenalty`: $\{ 0, 1, 2, 3, 5, 7, 10, 20, 50 \}$

This yields 21 600 parameter sets in total, but not all of them finished in the 20 hour limit that we set for the jobs. We have to note that for the number of jobs sent to the cluster to be manageable, each of the jobs had the first 4 parameters from the list above set and was testing all the 9 possible values for the DFPMP parameter. Therefore, it is possible that some parameter sets that would have finished on their own in the 20/9 ($= 2.\overline{2}$) hours time limit were lost because another setting took too long, but we do not think that this loss was significant for the overall results.

[1] `https://gitlab.com/gamedev-cuni-cz/theses/super-mario-astar/-/tree/master/Mario-AI-Framework/levels/test`

Out of the 21 600 parameter sets, 15 696 finished. Out of the finished ones, none of them has the TTFW value of 0, which means that no parameter set test with TTFW of 0 finished. This is understandable, as without the *run to the right* part of the heuristic, the search was not directed towards the goal enough. It can also be seen as losing the heuristic part of A*, which makes it work as a BFS (Breadth-first search) algorithm, and that apparently could not manage to solve the levels fast enough, even with the grid path heuristic.

The metrics we collected are the same as for the `MFF A*` agent. Figure 6.4 show the correlation matrix of the parameters and the metrics. We can notice that as opposed to `MFF A*`'s correlation matrix, the correlation of parameters with other parameters (especially NDW and TTFW) is not 0. This is caused by the fact that not all of the parameter set tests finished.

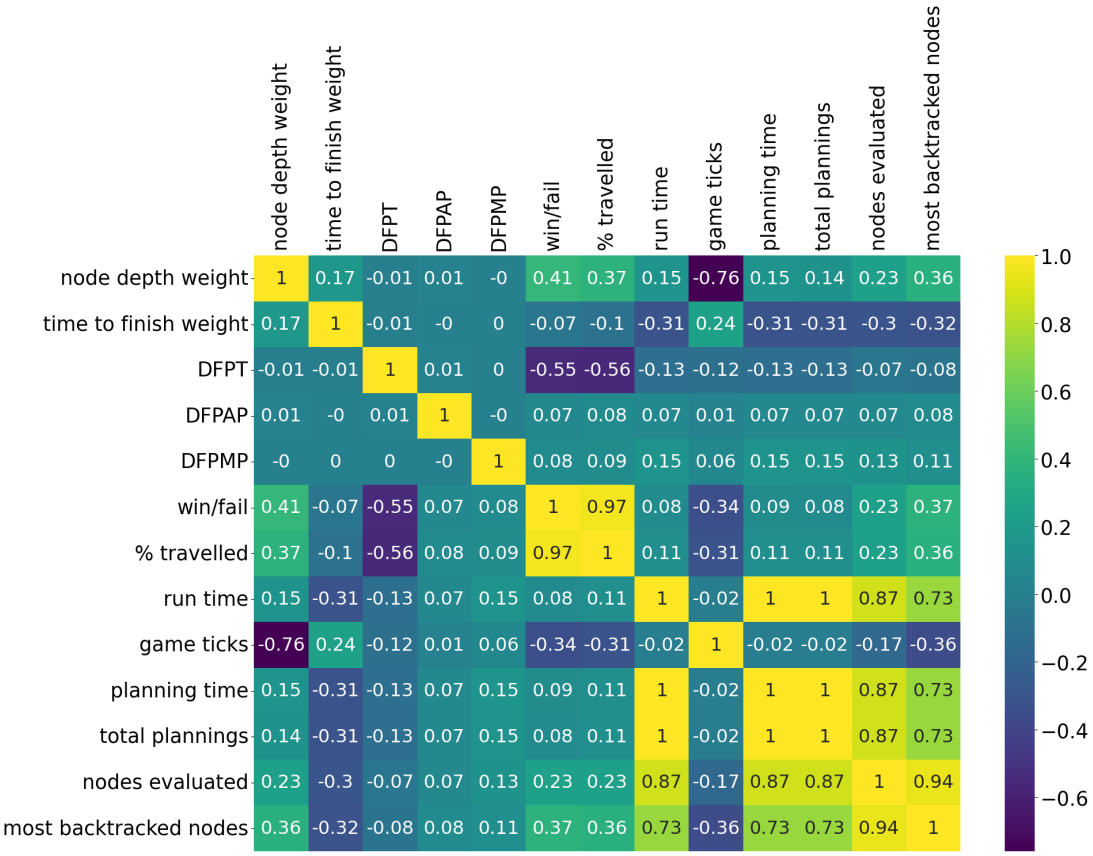


Figure 6.4: Correlation matrix of all the parameters and metrics, across all levels and all parameter sets.

6.4.1 Win rate and distance travelled

The most important thing to say is that many of the parameter sets reached a win rate of 100%. This means that we have successfully solved all of the test levels. As a matter of fact, 119 parameter sets reached 100% win rate.

If we look at which parameters correlate with win rate the most, we can see that NDW positively correlates with it and that DFPT negatively correlates with it. As to why there is a negative correlation with DFPT, the problem is that when we set DFPT too high, we are effectively removing this part of the cost function

(because the simulated width of the path corridor is so wide that it loses its informational value) and thus we lose the benefit of the grid path heuristic.

As for the positive correlation with NDW, it still applies that because of the implementation details, the value of NDW is effectively one-third of the parameter value, therefore the closer the value gets to 3.0, the closer it effectively gets to 1.0, and as the data show us, this is apparently the way to go.

Distance travelled, also noted as `% travelled`, is extremely correlated with win rate, as it denotes the average distance ratio that the agent travelled between a level's start and goal. Therefore, the same observation about the influence of NDW and DFPT applies.

6.4.2 Only won levels

As we have 119 agent versions that won all of the test levels, we have decided to do the metrics analysis only on these levels, as their data are not skewed by failing levels.

The reason why failing levels can skew the data is that if the agent fails at the very start of a level, its run time is recorded as really low, even though it did not achieve anything. Or the agent can get stuck and time out, which skews the run time in the opposite direction. And these are only a few examples of why the data of agents that won everything are more reliable.

Let us first look at the correlation matrix after filtering out all agents that did not win all levels. The matrix is shown in Figure 6.5.

We will now discuss the remaining metrics and try to explain why they may be correlated to some of the parameters.

6.4.3 Run time, planning time, total plannings, nodes evaluated, most backtracked nodes

As all of these metrics strongly correlate, we will discuss them together. These metrics have a very small positive correlation with NDW and a strong positive correlation with DFPAP.

For run time and planning time (which basically equal, as the overhead is very small) the fact that they increase with increasing DFPAP can be easily explained. When the agent follows the grid path, it is rewarded for staying within the DFPT (which, by the way, is never more than 4 for the agents that won all levels and is mostly equal to 1). But sometimes, staying on the path is not possible, usually because the agent needs to dodge an enemy. This necessarily means leaving the path to proceed further, but with a high DFPAP penalty, the search is immediately strongly penalized for trying to do so.

Therefore, the agent needs to explore a lot of nodes that stay on the path and inevitably fail before it finally gets to continue (leaving the path temporarily). As we just mentioned that the agents need to explore a lot of nodes before proceeding, this also explains the positive correlation between DFPAP and nodes evaluated.

The explanation for why run time, planning time and nodes evaluated slightly increase with increasing NDW might be that as NDW increases, TTFW becomes weaker in comparison, and therefore there is less drive for the agent to prefer

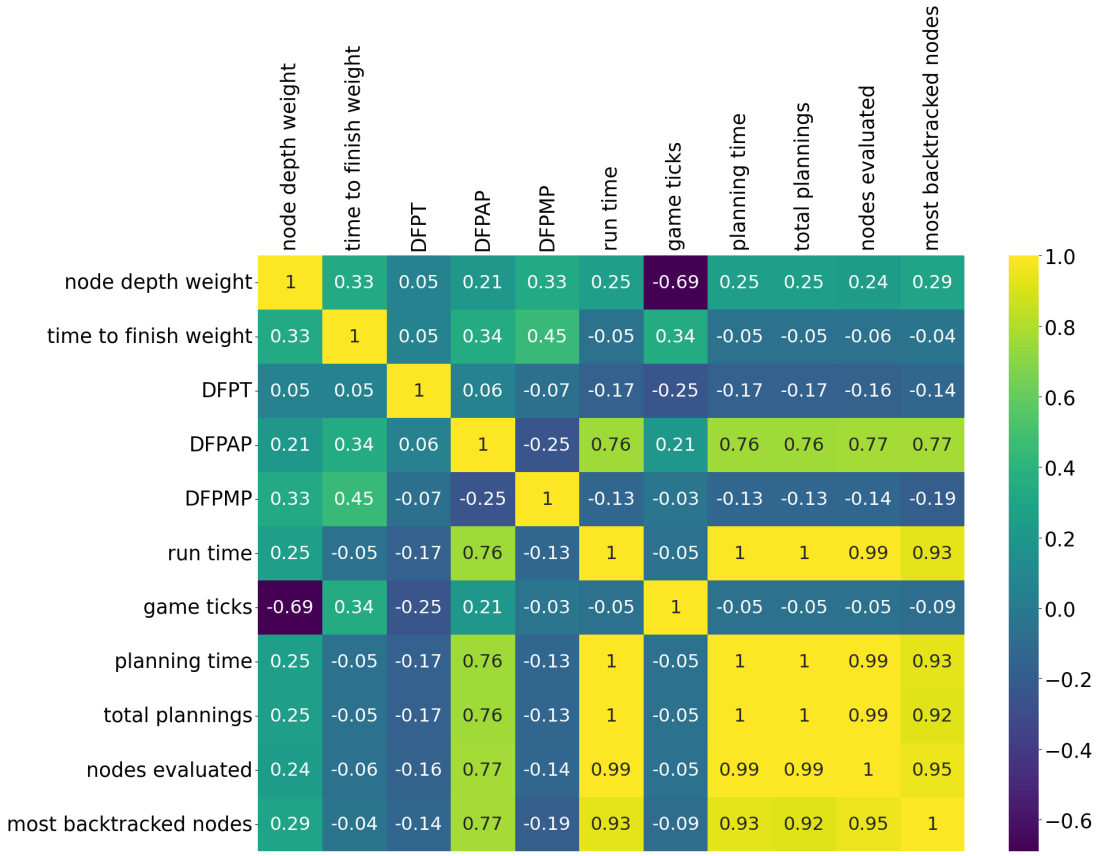


Figure 6.5: Correlation matrix of the parameters and the metrics, across all levels, but only for the parameter sets that won all levels.

states closer to the goal state. But this is just a speculation and the NDW correlation is rather small.

For the correlation of total plannings and DFPAP, we would say that if it takes the agent more time to find the goal state, it may not find it in the time allocated and so it has to start a new search next tick, thus increasing the number of total plannings.

Most backtracked nodes metric is probably correlated with DFPAP for the very same reason as all of the other metrics – as the agent might struggle to temporarily leave the path (to reach further states), it explores a lot of nodes before reaching a new farthest position, thus increasing the most backtracked nodes metric.

6.4.4 Game ticks

The game ticks metric is strongly negatively correlated with NDW and weakly positively correlated with TTFW. As the game ticks metric represents the time that an agent needs to reach the goal, lower values mean shorter paths.

It is expected that the average amount of game ticks needed to finish a level will decrease as NDW increases because the NDW parameter influences exactly the weight of the number of game ticks that we have taken so far to reach a state. Therefore, if this part of the cost function is more important, the paths found will be closer to optimal ones.

At the same time, if the TTFW parameter increases, the search is more biased towards states that are closer to the goal, not taking the number of game ticks required to reach the state into account that strongly. In other words, by using the TTFW parameter higher than NDW, we are using a weighted version of the A* algorithm, which may produce non-optimal paths.

6.4.5 Representative agents

Finally, we need to select the parameter sets that will represent the `MFF A* Grid` agent archetype in the comparison with other agents.

We have decided to include 3 agents in the final comparison. All of them have a 100% win rate and they were selected to represent various amounts of success when it comes to effectively exploring the game tree, which is represented by the number of nodes explored.

Therefore, the 3 selected agent configurations are:

- `Best Nodes Evaluated (BND)` – the agent with the least average amount of nodes evaluated (14 408). Parameter values: NDW: 1, TTFW: 2, DFPT: 1, DFPAP: 5, DFPMP: 5.
- `Worst Nodes Evaluated (WND)` – the agent with the largest amount of nodes evaluated (214 830). Parameter values: NDW: 1, TTFW: 0.5, DFPT: 1, DFPAP: 20, DFPMP: 1.
- `Average Nodes Evaluated (AND)` – the agent with the mean amount of nodes evaluated (33 919 – the 60th agent of the 119 that won all levels when ordered by nodes evaluated). Parameter values: NDW: 1, TTFW: 1, DFPT: 1, DFPAP: 10, DFPMP: 1.

Please note that the `BND` agent is also the 4th best one in terms of run time out of the agents with 100% win rate.

6.5 Agent comparison

In this section, we are going to compare the new agents with the previous state-of-the-art one [3]. We will be comparing the following 5 agents:

- `MFF A* 1,1.1` – the previous state-of-the-art agent with NDW of 1.0 and TTFW of 1.1, retested on the same levels as all of the other agents.
- `MFF A* 4,1.2` – the best agent from the `MFF A*` parameter search 6.2.
- `BND`, `WND`, `AND` – the representative agents for the `MFF A* Grid` archetype selected in Section 6.4.5.

We will first compare the agents on all of the 1015 test levels, then we will compare them only on the `krys` level pack, which we find the most interesting, and finally, we will compare them only on the levels that the `MFF A* 1,1.1` won. We will only be comparing their win rate, run time and nodes evaluated metrics, as those are the most interesting ones and the rest of the metrics mostly behaves according to them (see the correlation matrices).

6.5.1 All levels

We have gathered the results of the 5 agents on all of the 1015 test levels that we are using. Table 6.1 shows the agents' win rates (total for all the levels), Table 6.2 shows the agents' total run times (summed for all of the levels) and the relative comparison with the previous state-of-the-art agent (`MFF A* 1,1.1`), Table 6.3 shows the same for average nodes evaluated (across all levels).

Win rate

Agent	Levels won (%)	Levels won (count)
BND	100%	1015
WND	100%	1015
AND	100%	1015
MFF A* 4,1.2	99.11%	1006
MFF A* 1,1.1	98.62%	1001

Table 6.1: The total win rates of the agents on all of the levels.

As we already know, many of the `MFF A* Grid` agent configurations won all of the levels. And even if the improvement over the win rate of the `MFF A* 1,1.1` agent is not that significant, we believe that the reliability of the `MFF A* Grid` agents is much higher. Also, this is probably the first time that an A* agent has solved all of these levels.

Run time

Agent	Total run time (s)	MFF A* 1,1.1 comparison
BND	288	44%
WND	3 991	613%
AND	620	95%
MFF A* 4,1.2	588	90%
MFF A* 1,1.1	651	100%

Table 6.2: Total run times (summed across all levels, in seconds) of the agents. The last column shows a comparison of the times with the previous state-of-the-art agent – `MFF A* 1,1.1`. The values have been rounded.

And while the improvement in the win rate might not seem that impressive, the improvement in the run time is definitely significant. The `BND` agent has solved all of the levels twice faster than any `MFF A*` agent could, even with its parameters optimized.

We can also see that the choice of the parameters for the `MFF A* Grid` agent is very important, as the run times of the three `MFF A* Grid` agents differ a lot.

Agent	Average nodes evaluated	MFF A* 1,1.1 comparison
BND	14 409	25%
WND	214 831	369%
AND	33 920	58%
MFF A* 4,1.2	58 769	101%
MFF A* 1,1.1	58 176	100%

Table 6.3: The average amount of nodes evaluated (across all levels) for the compared agents. The last column shows a comparison with the previous state-of-the-art agent – `MFF A* 1,1.1`. The results are rounded.

Nodes evaluated

In terms of nodes evaluated, we can see that the `MFF A*` agents are four times worse than the `BND` agent, as opposed to two times worse run times. This difference will get even bigger when comparing only at more complex (`krys`) levels.

It is important to keep in mind that the metric is a bit influenced by the fact that the agents did not win all of the levels, and we will see how this changes when comparing only on levels that the `MFF A*` agents won.

6.5.2 `krys` levels

We consider the `krys` level pack to be the greatest challenge when it comes to navigating it efficiently. Because of that, we present the same comparison as for all of the levels, but only on levels from this level pack.

But instead of using the same `MFF A* Grid` configurations as for all levels, we found the best, worst and average parameter set of the agent for the `krys` level pack (out of those that won all of the levels). The configurations are as follows:

- `BND-krys` – the agent with the least average amount of nodes evaluated on the `krys` level pack (21 319). Parameter values: NDW: 0, TTFW: 1.5, DFPT: 0, DFPAP: 2, DFPMP: 2.
- `WND-krys` – the agent with the largest amount of nodes evaluated on the `krys` level pack (530 526). Parameter values: NDW: 3, TTFW: 1, DFPT: 5, DFPAP: 3, DFPMP: 1.
- `AND-krys` – the agent with the mean amount of nodes evaluated on the `krys` level pack (236 128). Parameter values: NDW: 0.5, TTFW: 2, DFPT: 0, DFPAP: 50, DFPMP: 0.

Table 6.4 shows the win rates, Table 6.5 shows the run times and Table 6.6 shows the nodes evaluated metric.

Win rate

Even though we consider the `krys` level pack to be the most interesting and hard for a human player, all of the agents have managed to beat it. But the interesting part is going to be the comparison of run times and nodes evaluated.

Agent	Levels won (%)	Levels won (count)
BND-krys	100%	100
WND-krys	100%	100
AND-krys	100%	100
MFF A* 4,1.2	100%	100
MFF A* 1,1.1	100%	100

Table 6.4: The agents' win rates on the `krys` level pack.

Run time

Agent	Total run time (s)	MFF A* 1,1.1 comparison
BND-krys	42	12%
WND-krys	876	253%
AND-krys	481	139%
MFF A* 4,1.2	358	103%
MFF A* 1,1.1	347	100%

Table 6.5: The agents' total run times on the `krys` level pack. The last column shows a comparison with the previous state-of-the-art agent – `MFF A* 1,1.1`. The results are rounded.

As foreshadowed, the `krys` levels are the most intricate to navigate efficiently, therefore we can see much higher differences in the performance of the agents.

The `BND-krys` agent is more than eight times more performant than the `MFF A*` agents, and the difference is even greater for the `AND-krys` and `WND-krys` agent, once again showcasing that the selection of the best parameters possible is critical.

We can also see that the `MFF A* 4,1.2` agent performed slightly worse than the `MFF A* 1,1.1` agent, most probably because its heuristic is weaker.

Nodes evaluated

Agent	Average nodes evaluated	MFF A* 1,1.1 comparison
BND-krys	21 320	7%
WND-krys	530 527	176%
AND-krys	236 129	78%
MFF A* 4,1.2	321 649	107%
MFF A* 1,1.1	301 549	100%

Table 6.6: The agents' average nodes evaluated on the `krys` level pack. The last column shows a comparison with the previous state-of-the-art agent – `MFF A* 1,1.1`. The results are rounded.

Finally, comparing the amounts of nodes evaluated, we see that the `BND-krys` agent (of the `MFF A* Grid` archetype) is a huge improvement over the `MFF A*` agent archetype, being 14–15 times more efficient in exploring the game tree.

6.5.3 Only won levels

As we have previously mentioned, comparing on levels where any of the agents failed might skew the results – in both ways. If the agent dies very early in the level, its run time and amount of nodes evaluated are recorded as really low; if it gets stuck, its run time and amount of nodes evaluated might be very high, skewing the results from other levels.

To address this issue, we present the agent comparison once more, this time only on levels that all of the agents won. To get these levels, it was sufficient to remove the levels that the `MFF A* 1,1.1` agent failed on. We have removed 4 `hopper` levels, 3 `ore` levels, 3 `patternCount` levels, 2 `patternOccur` levels and 2 `patternWeightCount` levels.

As presenting the win rates makes no sense, since they all are 100%, let us look just at the run times and nodes evaluated. Also, let us only compare the `BND` agent with the two `MFF A*`-based ones, because we are mainly checking the consistency of the results for all levels. The result can be seen in Tables 6.7 and 6.8.

Run times

Agent	Total run time (s)	MFF A* 1,1.1 comparison
BND	256	50%
MFF A* 4,1.2	425	82%
MFF A* 1,1.1	515	100%

Table 6.7: The total run times of the agents on levels that the `MFF A* 1,1.1` agent won. The results are rounded.

The `BND` agent still outperforms the `MFF A*` agents quite significantly, but we can see that the difference is a bit smaller, therefore the scenario where the `MFF A*` agent got stuck on its failed levels probably occurred and thus skewed the agent's run times a bit towards higher values.

Nodes evaluated

Agent	Average nodes evaluated	MFF A* 1,1.1 comparison
BND	12 825	28%
MFF A* 4,1.2	39 787	86%
MFF A* 1,1.1	46 233	100%

Table 6.8: The average nodes evaluated of the agents on levels that the `MFF A* 1,1.1` agent won. The results are rounded.

For nodes evaluated, we can see the same trend as for the run times – the difference between the `BND` agent and the `MFF A*` agents is a bit smaller, but still significant.

6.6 The most universal agent

We have established that the `BND` parameter configuration of the `MFF A* Grid` agent is the best one in terms of nodes evaluated – out of the agents that won all levels (it is also the 4th best in terms of run time).

But this was evaluated based on metrics over all of the levels together. Thus, it may happen that the agent is actually quite bad for some level type and it makes up for it by being really good for a different level type.

For this reason, we wanted to find an agent that would be somewhat good for every level type. We have decided to focus on the performance in terms of nodes evaluated, dismissing the win rate, because all of the agents have really good win rates and we want to find an agent that is overall the most performant. Also searching only among agents that won all levels would be a bit problematic, as we will explain later.

We look for the most universal agent by first finding the best agent in terms of nodes evaluated for every level pack. That means that for every level pack, we found the parameters with which the `MFF A* Grid` agent had the lowest nodes evaluated score. Table 6.9 shows the parameters and the average nodes evaluated for the best configuration found for each level pack.

Level pack	**Best parameters**	**Nodes evaluated**
ge	0.0, 2.0, 1.0, 0.0, 3.0	130.19
hopper	2.0, 2.0, 2.0, 3.0, 10.0	145.41
krys	0.0, 2.0, 1.0, 5.0, 3.0	19 093.65
Mario	0.5, 2.0, 1.0, 0.0, 2.0	364.00
notch	1.0, 2.0, 1.0, 0.0, 5.0	155.25
notchParam	1.0, 1.5, 2.0, 0.0, 20.0	159.16
notchParamRand	0.5, 1.5, 2.0, 5.0, 10.0	150.46
ore	1.0, 2.0, 0.0, 0.0, 3.0	307.33
patternCount	3.0, 2.0, 3.0, 10.0, 3.0	3 568.66
patternOccur	1.5, 1.0, 3.0, 1.0, 10.0	7 273.66
patternWeightCount	0.5, 2.0, 1.0, 0.0, 7.0	313.93

Table 6.9: For each level pack, this table shows which parameter set of the `MFF A* Grid` agent achieved the lowest average amount of nodes evaluated and the amount itself.

To obtain the most universal agent, we did the following for every parameter set of `MFF A* Grid` – for each configuration, we calculated the mean square error between its nodes evaluated metric and the best nodes evaluated result that we found, for each level pack. We then averaged the resulting mean square errors.

The parameter set which achieved the lowest average mean square error could be proclaimed the most universal parameter configuration, as there is no level pack where it would perform significantly poorly. It is the following parameter set:

- NDW: 0.0, TTFW: 1.0, DFPT: 0.0, DFPAP: 0.0, DFPMP: 2.0.

The search for the most universal parameter set was not restricted to parameter sets that won all levels, as this would make the search problematic. An agent

that had the best nodes evaluated score on a given level pack might not have won all levels of a different level pack, so the search would not make sense.

Let us call this *most universal agent* with the found parameter as `MFF A* Grid Universal` (`Universal` for short).

But before we proclaim this agent as the best choice for testing levels with previously unknown properties, let us look at Table 6.10, which shows the agent's average nodes evaluated per level pack compared to the `BND` agent, and at Table 6.11, which shows the agent's performance on all of the levels compared with the `BND` agent.

Taking all of these data into consideration, we would choose the `BND` agent over the `Universal` one, but it is up to every user of our agents to make their pick.

Level pack	**Universal - NE**	**BND - NE**	**Ratio**
ge	6 963	372	1 871%
hopper	723	179	403%
krys	31 164	91 394	34%
Mario	7 072	13 797	51%
notch	191	171	111%
notchParam	190	185	102%
notchParamRand	166	172	96%
ore	6 414	2 058	311%
patternCount	15 412	10 306	149%
patternOccur	44 856	38 981	115%
patternWeightCount	6 960	358	1 939%

Table 6.10: This table shows the average amount of nodes evaluated per level pack for the `Universal` and the `BND` agent. The `Ratio` column shows a relative comparison of the agents on each level pack, computed as (Universal - NE / BND - NE) * 100. The values are rounded.

Agent	**Win rate**	**Run time**	**Nodes evaluated**
`Universal`	98.82%	567.76	11 244.47
BND	100%	283.91	14 408.73

Table 6.11: Overall comparison of the agents' win rates, total run times and average nodes evaluated.

Please note that the `Universal` agent might have fewer nodes evaluated in Table 6.11 because it did not win all levels. It is actually the 8th best agent in terms of nodes evaluated (from all of the parameter sets, including configurations that did not win all levels).

7. Related works

Mario AI framework, Robin Baumgarten's agent, our previous work

The first version of the Mario AI framework was created by Karakovskiy and Togelius [11]. After 10 years, it was improved by Khalifa [12]. In our previous works we created a new forward model for the framework [1] and also new state-of-the-art artificial agents for Super Mario Bros. [3].

Works dealing with A* improvements

A paper by Chen and Sturtevant [13] presents various priority functions, while Foead et al. [14] did a literature review of A* pathfinding – we have gathered a lot of ideas from both of these papers, e.g. to thoroughly experiment with how different parts of an A* cost function are weighted.

We did not find any papers dealing specifically with A* improvements for platformer games.

Works using Super Mario agents

In this section, we want to demonstrate why our work in creating better artificial agents for Super Mario Bros. is meaningful. We will list a few works using such agents and describe what the work is about and how it uses an artificial agent.

Please note that Robin Baumgarten's agent is the best agent of the 2009 Mario AI Competition [15], which has since been used in many works – and we have significantly outperformed this agent in our previous work [3] and even more in this one.

Before we get to the list of a few selected works, we would like to mention that the literature review that we did for our previous work [3] found 29 Super Mario articles working with A* explicitly and 13 articles using the Robin Baumgarten's agent. The list of works using Super Mario Bros. artificial agents includes:

- **Intentional Computational Level Design (2019)** [16] – Super Mario level generation using evolutionary algorithms; uses Robin Baumgarten's agent as a model of a perfect agent to evaluate level segments.
- **Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network (2018)** [17] – level generation using GANs; uses Robin Baumgarten's agent to determine the playability of a given level.
- **Super Mario as a String: Platformer Level Generation Via LSTMs (2016)** [5] – generation of levels using LSTMs; uses a closer unspecified *tile-level A* agent* to gather path information for LSTM training.
- **Mario Level Generation From Mechanics Using Scene Stitching (2020)** [7] – level generation by composing pre-generated scenes; the scenes are labelled with the mechanics that Robin Baumgarten's agent triggers while playing them.

8. Conclusion

Our goal was to optimize the game tree search for the Super Mario Bros. game, which should allow us to create new state-of-the-art agents.

We started by analyzing the challenging types of levels and coming up with different ideas of how to solve them. During the analysis, we also noticed that some of the levels were not solvable, therefore our work also includes fixing such levels. After the analysis, we implemented several new artificial agents.

First, we have improved the existing `MFF A*` agent by tuning its parameters, which yielded the `MFF A* 4,1.2` agent that beat more levels that the previous state-of-the-art agent of this archetype. Then, we implemented the grid search and used the grid paths that were now available to us to create a new agent archetype – `MFF A* Grid`, which utilizes the grid path to navigate levels more efficiently. The parameters of the new agent were also tuned by searching for the best parameter set using the MetaCentrum cluster.

Finally, we selected the best parameter sets for both agent archetypes and compared them in various scenarios, both between each other and also with the previous state-of-the-art agent. Based on the evaluation, we can confidently say that we managed to create new state-of-the-art agents that significantly outperform the previous ones and which should now be the best choice for any Super Mario Bros. work (e.g. level generation) that requires level playability evaluation. The best agent is the `MFF A* Grid` agent with the `BND` parameter configuration.

Future works may include either further improvement of the agents, where we still see more potential in solving maze-like levels, or the creation of agents for games similar to Super Mario Bros. (2D platformers), potentially trying to come up with a more general agent architecture that could easily be used for a wider range of such games.

Bibliography

[1] David Šovald. *Efficient forward model for Super Mario AI framework*. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwaru a výuky informatiky, 2021. URL `http://hdl.handle.net/20.500.11956/127957`.

[2] Julian Togelius, Sergey Karakovskiy, Jan Koutnik, and Jurgen Schmidhuber. Super Mario Evolution, 2009. URL `http://julian.togelius.com/Togelius2009Super.pdf`.

[3] David Šosvald, Michal Töpfer, Jan Holan, Vojtěch Černý, and Jakub Gemrot. Super Mario A-Star Agent Revisited. In *International Conference on Tools with Artificial Intelligence, ICTAI*, volume 2021-November, pages 1008–1012, 2021.

[4] Frederik Schubert, Maren Awiszus, and Bodo Rosenhahn. TOAD-GAN: A Flexible Framework for Few-Shot Level Generation in Token-Based Games. *IEEE Transactions on Games*, 14(2):284–293, 2022.

[5] Adam Summerville and Michael Mateas. Super Mario as a String: Platformer Level Generation Via LSTMs. *First International Conference of DiGRA and FDG*, 2016.

[6] Eduardo Hauck and Claus Aranha. Automatic Generation of Super Mario Levels via Graph Grammars. In *2020 IEEE Conference on Games (CoG)*, pages 297–304, 2020.

[7] Michael Cerny Green, Luvneesh Mugrai, Ahmed Khalifa, and Julian Togelius. Mario Level Generation From Mechanics Using Scene Stitching. In *2020 IEEE Conference on Games (CoG)*, pages 49–56, 2020.

[8] Stuart J. Russell and Peter Norvig. *Artificial intelligence: A Modern Approach*. Pearson Education, Harlow, 4rd edition, 2020. ISBN 978-1-29202-420-2.

[9] GNU. Free Software Foundation. Bash [Unix shell program]. URL `https://www.gnu.org/software/bash/`.

[10] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42. Association for Computing Machinery, 2018. URL `https://doi.org/10.1145/3185768.3185771`.

[11] Sergey Karakovskiy and Julian Togelius. The Mario AI Benchmark and Competitions, 2012. URL `http://julian.togelius.com/Karakovskiy2012The.pdf`.

[12] Ahmed Khalifa. Mario AI Framework, 2019. URL `https://github.com/amidos2006/Mario-AI-Framework`.

[13] Jingwei Chen and Nathan R. Sturtevant. Necessary and Sufficient Conditions for Avoiding Reopenings in Best First Suboptimal Search with General Bounding Functions. *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021. URL https://doi.org/10.1609/aaai.v35i5.16485.

[14] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. A Systematic Literature Review of A* Pathfinding. *Procedia Computer Science*, 2021. URL https://doi.org/10.1016/j.procs.2021.01.034.

[15] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. The 2009 Mario AI Competition, 2010. URL http://julian.togelius.com/Togelius2010The.pdf.

[16] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional Computational Level Design, 2019. URL https://arxiv.org/pdf/1904.08972.pdf.

[17] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon Lucas, Adam Smith, and Sebastian Risi. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network, 2018. URL https://arxiv.org/pdf/1805.00728.pdf.

List of Figures

List of Tables

List of Listings