Human Achievable Path Generation in Video Games Through Modified Heuristic Search

by © Robert W Bishop Thesis submitted

to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Science, Department of Computer Science, Faculty of Science

Memorial University of Newfoundland

November, 2022

St. John's Newfoundland and Labrador

Abstract

We present a series of parameterizable modifications to heuristic evaluation of actions in the A* algorithm, designed to create human achievable and dexterity-robust paths through games in the 2 dimensional platformer style. We attempt to create paths designed for various levels of player skill by imposing constraints onto the timing and duration of actions in such a way as to mimic human reaction times and ability. We show that these modifications result in the A* search algorithm producing smoother paths, taking safer routes to avoid danger, and requiring fewer actions to be performed in a given amount of game time.

Acknowledgements

I would like to thank Doctor Churchill for encouraging my education and guiding me through the years. Additionally I thank my family for their support, and my partner Melissa for keeping me as human-like as possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

2-Dimensional platforming video games (platformers) have been a popular genre of video game for many decades, with the most popular example being the Super Mario Bros. franchise of games launched in the early 1980s by Nintendo. In these games, players control an animated character that can run and jump throughout an environment and do battle with enemies in order to reach the end of a given level. Platformers typically vary in difficulty of play, with harder games strategically placing level geometry such that running and jumping require very specific input timings around dangerous obstacles such as enemies or pitfalls.

Modern pathfinding algorithms have been shown to be able to produce paths through levels in these games, with some particularly popular examples being the Infinite Mario AI and MarIO neural network, which have garnered millions of views online. These videos are popular not only for the impressiveness of the technical challenge, but also because the resulting play is beyond the skill level that is possible by human players. The paths created by these systems are visually impressive — flying through levels, making frame-perfect jumps, and narrowly avoiding danger by mere pixels. Impressive as they are, they are only made possible by the perfect input precision of a computer AI player, and if a human player attempted to follow the same paths, any minor mistake would lead them to certain doom. In the video game speedrunning community this type of computer aided play is called a *Tool-Assisted Speedrun* or TAS[1], and paths that are only possible via superhuman input precision are called "TAS-only" strategies — while technically possible, they are to implement for a human trying to learn to play the game.

With the introduction of *New Super Mario Bros Wii* in 2009, Nintendo unveiled their *Super Guide* feature, intended to help struggling players. This mode restarts the current level and hands control of the player over to a pre-recorded instance of another human playing the level. This system is intended to serve as a tutorial for new players, reducing frustration by giving tips on how to accomplish goals within the game. While this feature is indeed helpful, the recording of human completions of a level require a significant time and budget cost to the game developers, and would have to be re-recorded whenever a change to the level is made. It would be advantageous to game developers offering these helpful features if the paths could be produced automatically by AI systems, rather than relying on human authorship.

1.1 Motivation

Existing AI pathfinding systems typically produce super-human paths that would be impossible to implement by humans, especially those just learning to play the game. These techniques use pathfinding whose action evaluations optimize paths for either distance or speed, with no care for the relative *difficulty* of emulating the resultant actions, owing to the fact that artificially controlled players have perfect input precision. We attempt to address these concerns by imposing a number of constraints onto the action evaluation function of the A* search algorithm. By paramaterizing these evaluations, we allow for a number of paths to be generated, with the goal being to generate helpful paths for players of differing skill levels. These paths are generated with consideration for the reaction time and dexterity of a human player, as well as the overall complexity of the path involved. This is accomplished by by limiting the frequency and number of actions a player would be expected to perform. In

[1]https://tasvideos.org/

addition to creating paths that are more human-viable, we examine the robustness of these paths to noise by perturbing the time at which the predetermined inputs are entered. By analyzing the success of these perturbed instances, we can determine how likely a theoretical path is to stand up to human re-creation, and even attempt to quantify a level of safety based on how likely these perturbed paths are to reach failure states.

1.2 Thesis Outline

Chapter 2 introduces definitions for the type of videogames we will be working with throughout the thesis, as well as a brief explanation of pathfinding algorithms as they apply to videogames generally. We will also discuss in depth several previous attempts at recreating human-like play in the field, as well as presenting a justification for our own original research.

Chapter 3 introduces our proposed modifications to the pathfinding algorithm we will be using to test our hypothesis, as well as the evaluation metrics we will be using to judge its ability to meet our requirements. A brief technical discussion of the programs, data structures, and datasets we have created and used is given as well.

Before discussing our experiments, we provide some visual analysis of our modifications in Chapter 4 with the intent both of providing the reader with a visual intuition, as well as demonstrating the method by which we selected a specific range of parameters to test.

Chapter 5 begins with a discussion of our experimental method before presenting the findings of our two large scale experiments. The first was conducted in order to find the best performing values for our modifications, which were then compared against eachother in a second experiment. The evaluation metrics discussed in Chapter 3 are presented for the data collected throughout, and discussed in terms of how they relate to our overall goal of human achievable path generation. Some additional metrics we chose to examine are included therein as well for completeness.

Finally we summarize our process and present an overall conclusion of our results in Chapter 6. We conclude with several proposed avenues for continued research in this field.

Chapter 2

Background and Related Work

Here we provide a definition for what we define as a platformer game, followed by an overview of pathfinding techniques for videogames. The A* algorithm will be discussed in detail, including how it is applied to real-time video games. We will overview how human performance and pathfinding have intersected in the past and examine how human-like play has been evaluated and benchmarked.

2.1 Platformers

The genre of videogames known as "Platformers" has a robust history, dating back nearly as far as the medium itself and still remaining popular today. "Despite all the improvements in graphics, hardware, and game design, platforming remains timeless", writes game critic Joshua Bycer in 2019 [2].

More accurately, this style of video game can be called a "Side scrolling 2 dimensional platformer". The verbosity of such a term means not all of these designations are included when discussing games in this genre; however it would serve us well to examine all these terms.

2 Dimensional: The game is played soley along two axes. The earliest videogames existed in two dimensions out of necessity, with early consoles like the NES only able to display a maximum of 64 sprites (two dimensional images), each consisting of 8x8 pixels [5].

Now the term is usually used to contrast the game to games played in 3d. A 3d game allows the player 3 axes of movement; however, many games in the modern age still elect to constrict their play to two dimensions of movement, even in cases where the game itself is constructed from an engine that allows for 3 dimensional rendering. Fighting games such as Street Fighter or Super Smash Brothers play this way, with the game visually being rendered using modern 3d techniques to incorporate depth and lighting, whereas the player experience is constrained to 2 dimensions, ie. the players can only move left, right, up and down.

Side Scrolling: In a side scrolling game the player is viewed from a side profile and the game world scrolls past them as the player moves as opposed to having a fixed or cinematic camera[27]. This is usually used as a contrast to “top down” games in which the player is seen from above. The perspective offered from a side scrolling game means the player is able to control their movement left and right, as well as up and down. Side scrolling games incorporation of verticality often means that some form of gravity is implemented allowing the player to run and jump.

Platformer: This term describes gameplay. Platformer games place an emphasis on controlling jumps around obstacles and onto platforms, often simply called “platforming”. Enemies may be present, but the challenge of the game usually lies in mastering the character’s movement rather than strategy or combat.

In order to further test the skill of players, many platformers emphasize reaction time through speed. This would become a cornerstone of the marketing for the Sega Genesis in particular their game Sonic The Hedgehog, considered a “Rival Franchise” to Super Mario, which was marketed explicitly as being faster, and implicitly, more fun than Mario [14]. “The definite impression of Sonic the Hedgehog is speed. Sonic moves so quickly that it is not always possible to perceive the environment” writes Jonathan Skyes in his essay “A player-centered approach to digital game design ”[23].

2.2 Pathfinding in Video Games

Here we will review some of the more popular pathfinding techniques used in videogames and discuss their different strengths and implementations, before discussing the A* algorithm specifically, and how it is applied to realtime videogames.

2.2.1 Methods

Steering Behaviours

Steering behaviors use compounding combinations of rules to determine the locomotion of autonomous agents and were pioneered by Craig Reynolds in the late 1990s [19]. These behaviours model agents as vehicles. These idealized vehicles have forward acceleration as well as a steering force able to change their rotation in much the same way as a real life vehicle. By interacting with the observable area around each agent, they use a series of predefined rules to determine which direction they should face, and how quickly they should move.

A basic example of steering can be seen in the "Seek" behavior. This attempts to steer the agent towards a specific location, continually aligning the agent's velocity vector such that it is aligned with the target. This simplistic behaviour can be improved to intercept moving targets by considering the targets respective velocity and predicting its future position. This behaviour, called "pursuit", may be simplistic, but is still used for basic projectile behaviour in some games to create "Homing" projectiles. An opposing behaviour called "flee" uses the same techniques but attempts to align the velocity away from a point, rather than toward it.

More complex behaviour emerges when these basic rules are combined. One such technique, obstacle avoidance, uses a combination of seek and flee behaviours. Through a combination of seeking a target and fleeing from boundaries labeled as obstacles, the resultant path can mimic seemingly complex behaviour.

Algorithm 1: Evaluating cost of cells c to destination D

```
Data: $C = \{c_1, c_1, \dots c_n\}, D = c \in C$
begin
    foreach $c \in C$ do
        $c.cost = \infty$
    $D.cost = 0$
    uneval = $\{c_1, c_1, \dots c_n\}$
    current = $D$
    while uneval $\neq \emptyset$ do
        foreach Neighbor $n$ of curr $\notin$ uneval do
            newcost = cost($c$, $n$)
            $n.cost$ = min(newcost, $n.cost$)
        uneval = uneval − current
        current = min(uneval)
```

Field Approaches

Rather than act directly on the game itself, these techniques create a grid based abstraction of the game space in order to derive their paths. In its most basic form the game space is abstracted into tiles, marked as walkable or not. A destination is selected, and the pathing information is computed for the entire grid, creating a 2 dimensional representation, a *field*, that can be used to guide agents to the goal.

Flow Fields also known as vector fields, are one such instance of this technique. Once the abstract grid has been created, Dijkstra's algorithm is used to calculate the total path cost for every node on the grid to reach. Once every cell's distance is calculated, a second pass is performed over the grid whereby every cell is assigned a direction vector pointing at its lowest cost neighbour. Once the grid is populated, the goal can be reached from any other point in the grid by simply following the direction vectors [6].

The flow field can be extended with additional information acting on the direction of the vectors. Such "influence maps" can represent abstract concepts such as enemy vision ranges,

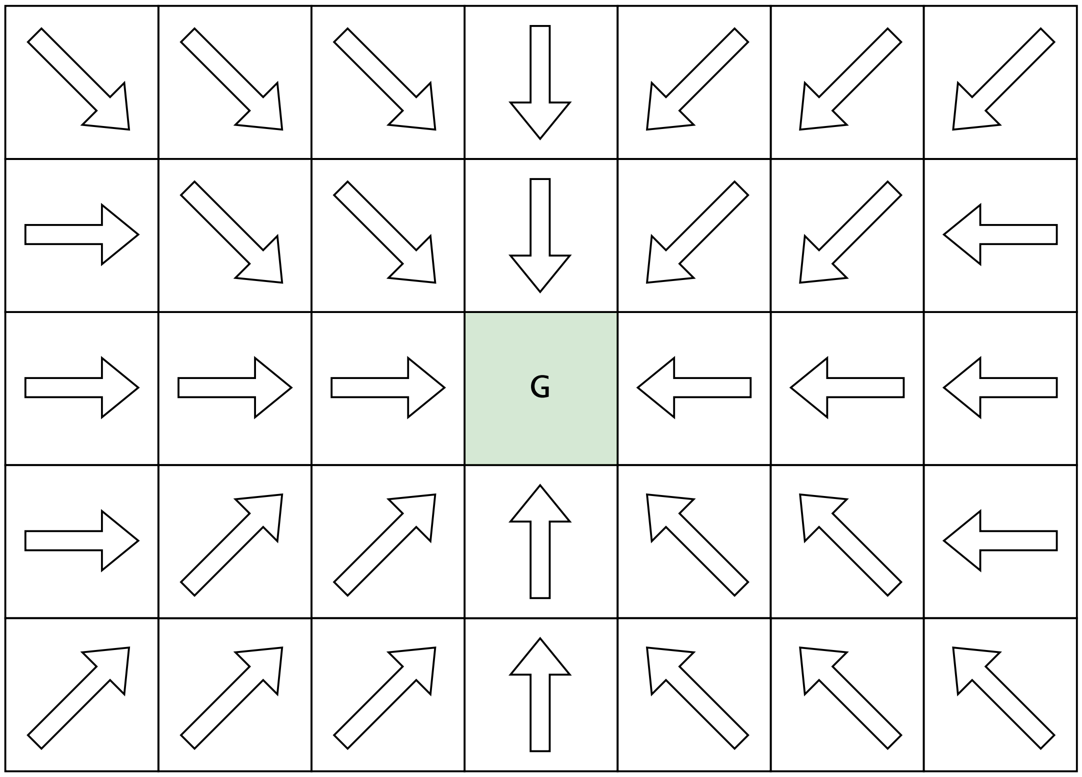


Figure 2.1: An example flow field. Following the arrows at any point will lead to the goal.

or points of interest. Several different such layers can be combined before the final vector field is calculated, leading to complex emergent behaviour.

Flow Fields excel at multi-agent pathfinding scenarios. Once the field has been computed for an area, multiple agents can path towards the goal with very little additional computation required. The upfront cost of computing the entire flow field however, often makes such approaches less than ideal for single agent pathfinding.

2.2.2 The A* Algorithm

A* is a popular and industry-wide algorithm for path finding in general cases [17]. Similar to Djkstra's algorithm seen in Algorithm 1, it successively calculates the path cost of cells between the start and goal points. Unlike Dijkstra's algorithm, however, it is designed to facilitate finding a single path between two specific points. This limitation is leveraged to improve the algorithm in ways that make it more suitable than Dijkstra's for the task of single-agent pathfinding.

Heuristic Function

Instead of simply calculating the cost required to reach all cells from the destination point, A* includes the addition of a *heuristic function* to estimate the cost from each cell to the end point. At each step of the iterative process, the cell that is selected to be evaluated is based on the sum of both it's cost as well as an estimated cost, calculated by a heuristic function. The final cost is then evaluated using the formula

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

where $g(n)$ is the cumulative cost of reaching the cell, and $h(n)$ is the evaluation of the heuristic. The final cost $f(n)$ then becomes an estimation of total path length. By always evaluating the cell with the lowest estimated path length, A* shows remarkable efficiency in practical pathfinding scenarios.

Graphs

A* (and Djkstras algorithm) do not operate simply on grid-based searches. Rather they operate more abstractly on graphs. Any grid based representation of cells as a pathable space can be expressed instead as a graph with nodes and edges, as seen in Figure 2.2. In the cases of abstract graphs, $g(n)$ and $h(n)$ must still be available and calculable. That is, we still require a way to calculate the cost between two nodes, as well as estimate the cost from a node to the destination. Cost does not always have to be distance. A* can be used, for example, to find a series of bus routes that minimize time to the destination, or minimize the literal cost of a series of plane trips.

Priority Queues

Because the selection process requires constant access to the node with the lowest f score, the implementation of how the nodes are stored is of particular importance, and a data structure known as a *priority queue* is typically used. A priority queue is an abstract data structure that operates on a set of elements, and can be implemented in many ways [4]. These implementations all share several key operations[1]

- **Insert(x, k):** Inserts element x into the queue with associated key value k.
- **Min():** Returns the element with the lowest key value.
- **Extract-Min():** Removes and returns the element with the lowest key value.
- **Decrease-Key(x, k):** Decreases the key value of element x to the value of k.

2.2.3 A* for Real-time Games

So long as methods exist to translate gameplay into a graph structure and calculate $f(n)$, we can turn a realtime game into a pathfinding problem we can solve with A*. In many modern implementations of real-time pathfinding, the level geometry is abstracted not to a grid, but to a series of differently sized nodes, known as a *navmesh* [10]. Navmeshes have proven very effective for 3d environments as well as for multi-agent pathfinding [29]. However the drawback is that they are themselves often difficult to compute, typically created along with the level environment, under the assumption that this environment remains static.

As an alternative to this approach, a graph can be directly created from the game's *state* and *actions*. In this approach the nodes are states, and the actions that can be taken at that point are the edges between them.

Game States are a representation of the entire game at a moment in time. This includes information about every entity on screen, their positions, velocities, and any additional information needed to recreate their behaviour. For the search algorithm to function correctly, the representation must be lossless. That is, whichever form of representation you use for a game state must be capable of perfectly recreating the game up to that point. As an example, consider algebraic notation in chess: the string of characters

[1]The Min-priority queue implementation is given as an example here.

1.Nf3 Nf6 2.c4 g6 3.Nc3 Bg7 4.d4 O-O 5.Bf4

is capable of perfectly recreating the board state of a game, despite not listing the positions of every single piece on the board at the time.

Actions are the inputs through which an agent may interact with a system. For guiding an agent through a platformer, this would typically correspond to actions such as “Move Right” or “Jump” though they can also represent autonomous agents in the world with other such actions, such as “Shoot at player”. For emulating human play there is often an advantage in that player inputs are themselves already perfectly discretized into actions: button presses. As such, the answer to the question “What actions can the player perform in this state?” is usually simply the buttons they can press.

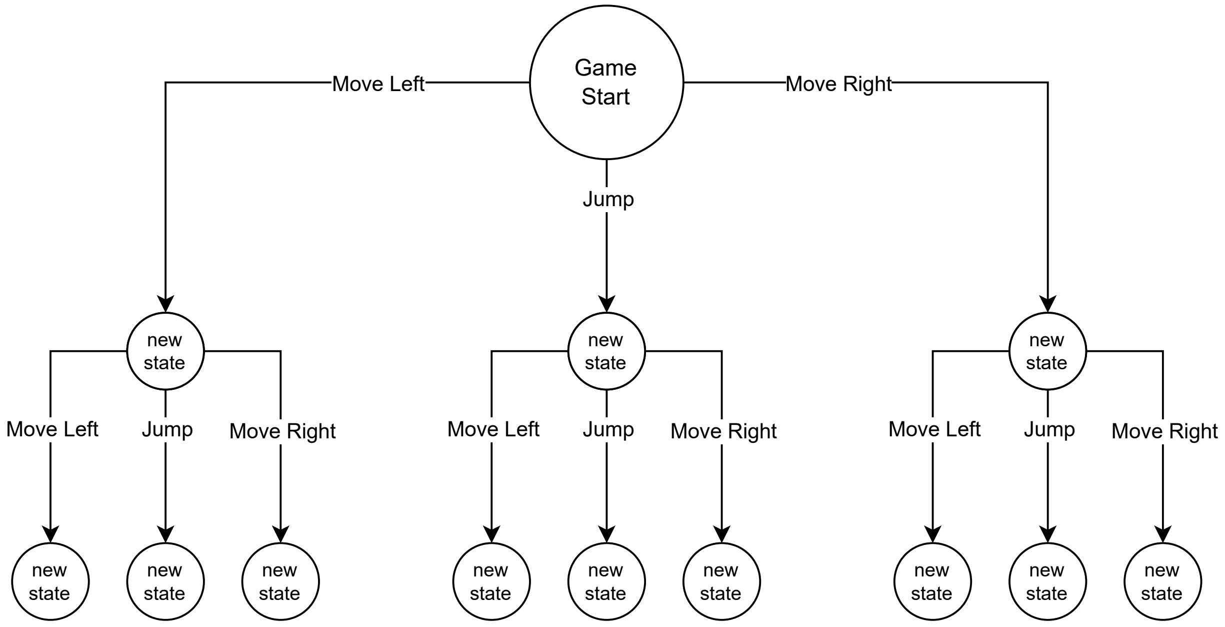


Figure 2.3: A graph created from a game where states are nodes and actions are edges.

To create this graph, the game needs to be *simulated* between all nodes. That is, once an action is selected, the game needs to advance to the following state obeying all the rules of the game as it would as if it were being played. The game also needs to be deterministic. If taking the same action from the same state does not always result in the same child state, then the A* algorithm will be unable to evaluate nodes properly.

With this in place, we can see the algorithm behind using A* in realtime games given in Algorithm 2. In this implementation A* can be used to optimize any definable goal, such as achieving a high score or staying alive as long as possible. However for consistency we will assume the goal is pathing the player to a point within the game world. $g(n)$ and $h(n)$ will therefore be based on distance as previously discussed. Instead of providing the algorithm with a grid based representation of the world, it is provided with the initial game state, a list of actions, and a destination condition. In our case, the destination condition will be that the player's position be within a certain distance of our goal point. This information must be retrievable from the game's state.

This algorithm produces a series of actions which, when performed sequentially in order from the initial game state, will provide the shortest path to the destination.

2.3 Realistic Platforming Agents

Human imitation has long been a goal in many aspects of computing [28], and video game AI is no exception. Attempts to mimic human play span nearly every genre, platform style games included. Here we will examine some of the most notable instances that apply to our research, as well as a brief discussion of how we add value to this space through our research.

In 2009 the Mario AI Competition was created [26] where several different implementations of Mario pathfinding AI were tested on procedurally generated platform levels in the style of Super Mario. In the first few years, A*-based controllers dominated the competition. No regard for realism or human ability was considered initially, however in the competition's final year of 2012, the Turing Test track was introduced [21]. As the name implies, contestants were given the goal of emulating human play, and were judged subjectively by a panel of observers. By this time no bots competing used any form of heuristic search, most preferring neural networks. The best performing bot in the Turing Track achieved a score of 25%, convincing less than half those surveyed that it was a human player.

The next year saw the release of a paper which researched the exact same concept: creating human-like play in Mario [12]. In this paper Ortega et al. compared several agents, including ones based on the Mario AI Turing Track. Every agent evaluated was either based on a neural network, reinforcement learning, or hard coded rules. At that time, it seems heuristic search algorithms were deemed insufficiently complex to emulate human play.

In 2016 Frydenberg et al. sought to use Monte Carlo Tree Search to emulate human behaviour, noting "in many cases, it is harder to create a human-like game-playing agent than to create a high-performing agent that wins many games or achieves superhuman scores" [7]. MCTS is a tree-search algorithm not unlike A*, but it bypasses the need for a heuristic function by using randomized playouts to determine the relative strength of a state. Their approach featured hard coded behaviors in order to facilitate branch pruning of "ineffective actions", and gave priority to certain higher level concepts such as a "Map Exploration Bonus". Unlike the previous examples, this was not focused on Mario, or even platform games in particular, instead using a subset of games from the General Video Game AI framework [13].

Fujii et al. sought to create what they called "Biological Agents" in 2013 [9]. They created two agents, one using Q-learning and one using A*, to play in the infinite Mario engine with a series of biological constraints. These include a perceptual delay, where the algorithm is given the character's coordinates with a slight delay in order to simulate human perception, as well as modeling physical fatigue. This resulted in an agent that would jump earlier than a typical A* implementation, as well as take short breaks, increasing its playtime by 128%. Analysis was done subjectively, through a survey of 20 participants, who were asked which seemed most skillful and which seemed most human-like.

2.4 Established Evaluation Metrics

Evaluating what makes something more human-like in many ways is the key difficulty behind emulating human behaviour via an algorithm. Human analysis through survey, as in the

Turing Test, is likely the best method despite its subjectivity. However without a clear set of rules or observations to be formalized it becomes rather difficult to implement into an algorithm in the first place.

A survey by Temsiririrkkul et al in 2017 attempted to create a classification system of AI behaviour by splitting all actions into four categories: inside the game, outside the game, related to the objective, and not directly related to the objective [25]. While interesting for generating novel play, it would not benefit the generation of paths to incorporate outside information such as empathy, or attempt to deviate from the main objective in our case.

Nevertheless, human comments on the subject prove to be an excellent starting point. Ortega et al. solicited comments as part of their Turing Test style evaluation, the top results of which were illuminating; in order, they were *Jitteriness, Useless moves, No long term planning, Too fast reaction time, and Overconfidence*

The 2013 Mario specific paper used a fitness score based on the traces of actual human playthroughs and applied this fitness as a heuristic to existing agents. While novel, it offers little utility in attempting to parameterize realistic pathing.

Phuc et. al used a similar approach in 2017 in their paper "Learning human-like behaviors using neuroevolution with statistical penalties" [16]. In this paper, they trained an agent based on the NEAT algorithm [3], a neural network that is capable of changing its structure through evolutionary methods over time. Similar to the previous method, they used a statistical sampling of real human data to inform a number of criteria, such as how often a player moves, how much time they spend in the air and how often they move left. Agents were penalized for failing to emulate play with similar statistical breakdowns of these features.

Ortega et al. included a number of evaluation criteria in their 2016 agent that were informative: Action Length, Nil-Action Length, and Action to New Action Change Frequency, paraphrased below:

- Action Length: How often a button is repeated, analogous to humans holding down buttons for more than one frame as opposed to agents who make single-frame actions.

- Nil-Action Length: How long the agent spends doing nothing, as humans often wait.

- Action to New Action Change Frequency: How often a player switches from one action to another.

The first and second metrics do not directly apply to creating paths as directly as the third, but all three were shown to influence the perception of believability of agents, and as such demonstrated that applying concepts of reaction time to rules-based systems is promising. Evaluation metrics such as these represent a step towards being able to systematize believability into objective functions, which can be incorporated into algorithms for generating believable agents.

2.5 Conclusions and Shortcomings

Heuristic search algorithms for videogames remain popular as a form of pathfinding in the modern era, and still represent a large field of research. While A* sees much use in the general case of shortest-path single-agent pathfinding, it has been much less closely associated with the small but growing field of human-like pathfinding. However, we believe none of the techniques surveyed are fully appropriate towards achieving our goal.

Our stated purpose is to create human-informed, parameterizable paths motivated by the idea that current autonomous pathfinding techniques produce paths that are generally unsuitable or downright impossible for humans to imitate. With this goal in mind, there are 3 significant shortcomings we have identified.

Intentional Flaws

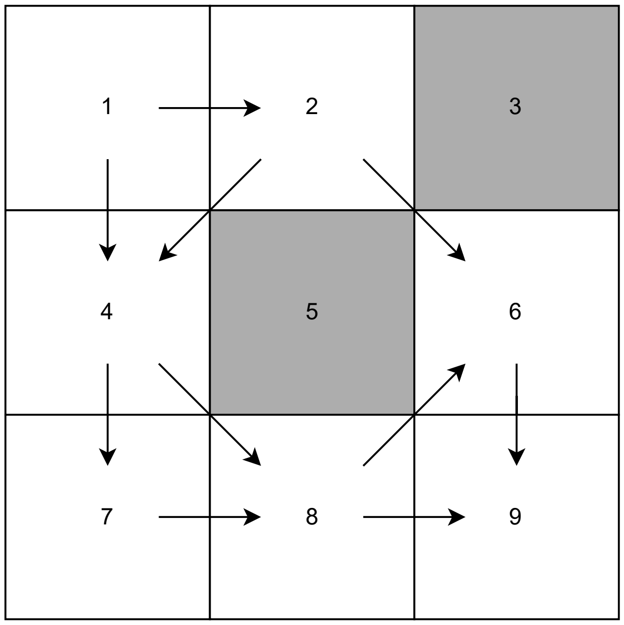
Most work towards developing human-like autonomous agents focuses on emulating imperfect human play. These agents are designed to make mistakes, hesitate, and other such actions, typically with the intention of fooling jurors into believing they are human. While we share many of the same goals as these agents, such as reducing jitteryness and superhuman reaction times, the techniques employed, such as training agents based on real life play or purposefully rewarding mistakes do not lead themselves toward the goal of generating practically useful paths.

Parameterization

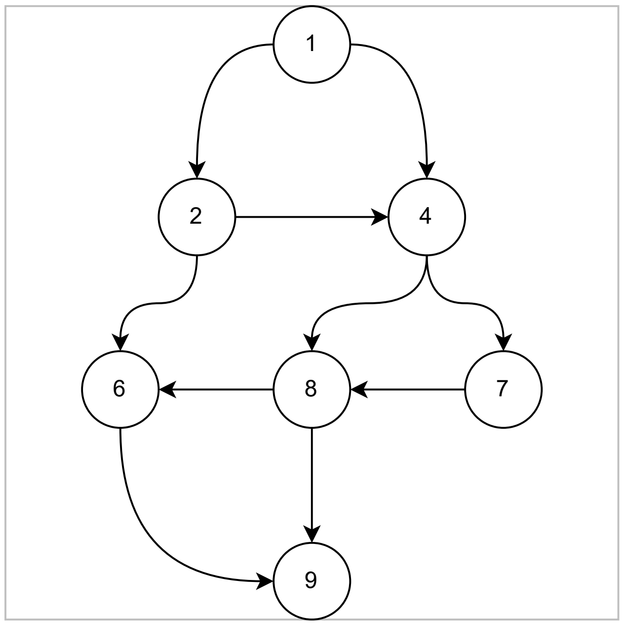
Parameterization of most of the methods surveyed is impractical or impossible. Neural network based agents would have to be re-trained to change any aspect of their play, a process known to be expensive in terms of computing time. Several of the agents involved were trained using datasets of real life human play. As such, new datasets would have to be acquired, and again the agent would have to be re-trained. Furthermore, any biases inherent to the training sets could potentially be introduced.

Human Benchmarks

While many of the agents focus on believability, very few reported on reaction time statistics or other such human benchmarks. It is possible that the paths they created may in fact involve subtle but unapparent movements making them more difficult to achieve than it appears. Additionally, little information is reported on the established metrics introduced in section 2.4. It would be useful to know to what extent a believable agent can reduce jitter, action length, action change frequency, etc.



(a) Possible paths from cells 1 to 9 (grey cells unpathable)



(b) The same grid expressed as a graph

Figure 2.2: Pathfinding over grids can be considered a graph problem

Algorithm 2: A* for realtime games

```
Input: Game State $S$, Destination Condition $D(S)$, Actions $A = [a_1, a_2, \ldots a_n]$,
       Cost Function $g(S)$, Heuristic Function $h(S)$
Given: Node class $N()$ which, when constructed from states creates new Node, $N_S$
begin
    Open = $[N]$                                          // Priority Queue
    Closed = []
    while Open is not empty do
        current = Open.Extract-Min()
        if $D$(current) then
            Return list of parent actions of current
        Add current to Closed
        foreach Action $a$ in $A$ do
            $S_a$ = current.simulate($a$)       // Create child state from action
            if $S_a$ in Closed then
                Continue
            $S_a.g = g(S_a)$
            $S_a.h = h(S_a)$
            $S_a.h = S_a.g + S_a.h$
            Add $N(S_a)$ to Open
    Return Fail                                           // No path exists
```

Chapter 3

Methodology

This section discusses our approach to implementing our proposed system. It begins with a discussion of the custom implemented 2 dimensional game engine, as well as the data structures we use to represent searches and paths. We continue with a discussion of the specific modifications we have made to the A* algorithm, called Action Value Modifiers (AVMs) as well as the metrics we have decided to use to evaluate the performance of said modifications.

3.1 The Environment

To best suit our needs we are working in a custom game engine designed specifically to create arbitrary 2d games. This game engine uses an Entity Component System (ECS) architecture, and is written entirely in C++ from scratch. The only external libraries used are the Simple Fast Multimedia Library (SFML) [1] which allows for the rendering of textures and handling of user input, as well as an external JSON parser to allow for easy real-time configuring of variables. The engine has several features that facilitate our research goals:

- It is completely deterministic
- It is capable of generating and reading replays to recreate play

[1] `https://www.sfml-dev.org`

- It is capable of quickly and easily copying game states to allow for easy integration into heuristic search algorithms
- At all times, the engine knows the actions available to the player, and these can be queried from outside functions.
- The rendering system can be disabled (headless) for maximum speed, achieving several thousand frames per second of simulation.

As stated above, the engine can be configured to run in a headless mode, wherein it takes a list of parameters from a configuration file and is capable of directly comparing an arbitrary number of searches in random locations across a corpus of levels. Levels themselves are loaded from text files, and conform to the format specified for the Video Game Level Corpus [22]. This can be seen in Figure 3.1 which some may recognize as the well known 1-1 level of the original Super Mario Brothers. For simplicity and ease of rendering, objects without collision are omitted.

The greatest advantage of using a custom system is that our game engine logic / data and search algorithm code retain a large degree of separation and modularity. The search algorithm receives all actions as well as positional data from the game engine, and explores by selecting an action, after which the simulation is run and the resulting state given back to the algorithm for evaluation. Many similar systems use hard coded values for their goals and heuristics: for example many Super Mario AIs have precomputed information about potential jump arcs, or use the player's x position as the goal, as the end goal of Mario is ultimately to run as far to the right as possible. These modified heuristics are based on assumptions about the game: for example the physics never changing, and the goal always being the furthest right point of the level. These assumptions can allow for a large degree of discretization for their specific task, but by not using any such assumptions our system retains more generalizability. For example, in our engine it is possible that mid-way through a search, gravity could be completely removed and the input methods changed to resemble a top-down game. Because the search only knows the list of possible legal actions and the

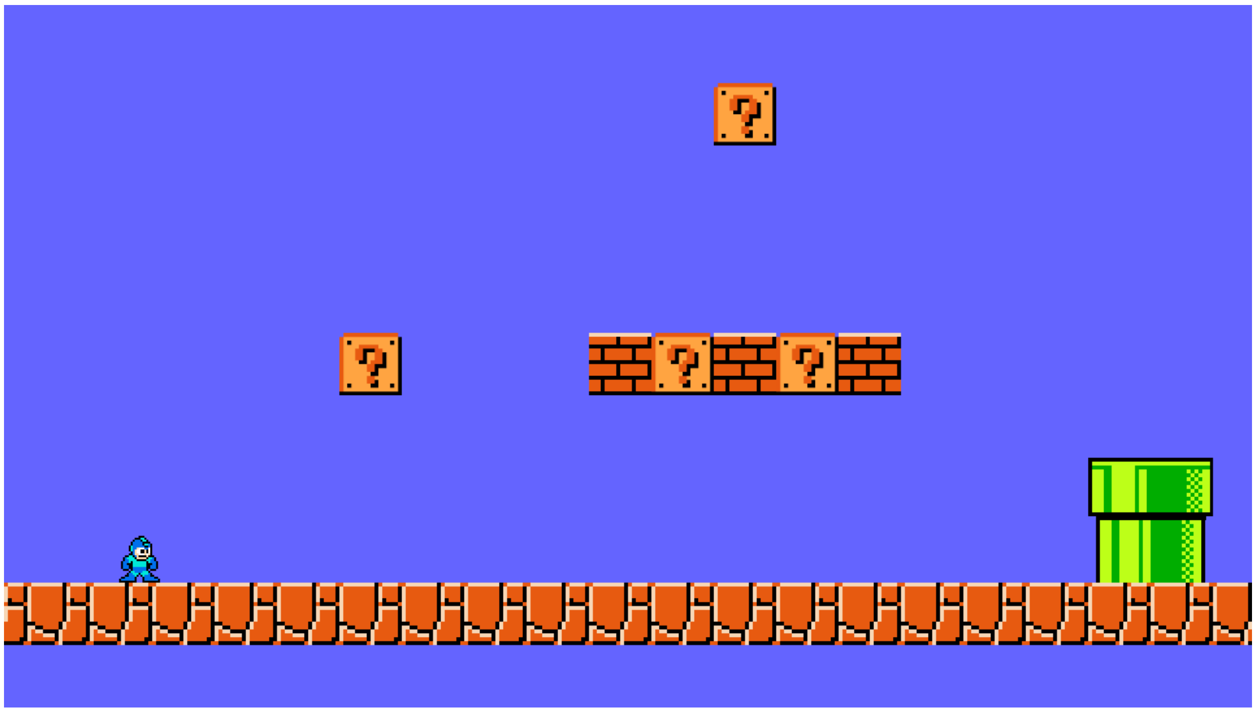


Figure 3.1: A screenshot of our engine. The player character is at the bottom left.

resultant position, reusing the system can easily be achieved on arbitrary games created within the engine.

3.2 Searches as Data

Central to our methodology are the concepts of *Searches* and *Replays*, the former being a class of functions which perform a search instance within a game state which generate the latter, which itself is simply a data class representing the actions performed by the search.

We define a Replay R as an ordered sequence of actions, A_1, A_2 ... A_n, with each action A_i consisting of two elements: a_t (the time at which the action occurred, measured in game frames), and a_n (the name of the action being performed, such as *JUMP*, *MOVE*, or *SHOOT*). Given that our engine is completely deterministic, this means that a sequence of play can be perfectly replicated from a replay, assuming the starting conditions of the game state are the same. Replays and states can be stored as text files for ease of debugging and

replay.txt
```
0    START-RIGHT

18   START-JUMP

30   END-JUMP

48   END-RIGHT
```

Figure 3.2: A sample replay file from our engine.

visualization. If at any time step in the game the AI system issues no inputs, it does this via a *NO-OP* action; however, we elect not to store the *NO-OP* action in replays as it can be assumed by default, and its inclusion would produce needlessly large replay files. Figure 3.2 gives a simple example of the contents of a replay file.

Action names are defined in the game engine as part of the logic of the game being simulated. Their meaning, while useful to humans, carries no semantic information to the algorithm. The performing of any complete action in our engine is broken down into two distinct steps: the start and end of that action. The start of an action such as running to the right: (*START-RIGHT*) would be analogous to a player pressing down on the button that moves the player to the right, while the *END-RIGHT* action would be the player releasing that button. This format was chosen to minimize the number of actions required in the replay file, as well as to allow the replay files to be human-readable.

A *search* represents a single instance of our A* search algorithm, where all of the action value modifiers are defined. An important aspect of the algorithm is the concept of frame skipping. Frame skipping is one of the standard techniques for game state exploration and pathfinding and has a documented history [1]. Rather than expand a new game state every frame, once an action has been decided, the search continues the simulation of the game for a number of additional frames before expanding to the next frame. Practically, this imposes a limit on how often the search can take actions, as once an action is taken it will continue uninterrupted for the next n frames. While Deepmind's Atari DQN network used a frame skip value of 4 [11], after experimental testing we opted to use a more lenient 6 frames initially. Given that our game engine runs at a consistent 60 frames per second (FPS) this also gives us a round value of 10 actions per second. This is an upper bound on the frequency of actions, and we have observed that the vast majority of replay files contain far fewer than 10 actions per second on average.

3.3 Action Value Modifiers

One of the core ideas of our research is that of the Action Value Modifiers (AVM), which act as modifiers on the heuristic cost function of nodes in our search algorithm by examining the timing and quality of actions. By modifying the A* search algorithm's heuristic evaluation we are effectively imposing constraints on the behaviour of the resulting paths. The result of these modifications is to attempting to prioritize the exploration of more human-friendly paths by imposing penalties on the cost of performing actions that we deem to be super-human, such as performing too many actions in a short duration of time. A description of each of these modifiers is as follows.

3.3.1 No-Op Modifier (NOM)

No-Op refers to the search choosing to take no action on the given frame. It is the most common decision, and does not correspond to taking an action by our definitions. Because stopping an action is itself an action, the No-OP action will leave the search actor performing whatever action(s) it was previously performing. It can be helpful to think of No-Op as "changing nothing" as opposed to simply "doing nothing". Continuing, any references to taking an action are assumed not to include No-Ops.

This modifier multiplies the value of the no-op action, and as such setting it to < 1.0 (incentivizing) causes the algorithm to favor paths with fewer actions. Because this action is by far the most common in normal human play of most video games, we theorize that setting the NOM to lower values will result in more human-like behaviour.

3.3.2 Action Change Modifier (ACM)

Essentially the inverse of the previous modifier, when set to a value higher than 1 it imposes a multiplicative penalty for every instance of taking an action. Penalizing these actions will attempt to find paths with fewer action changes, which results in smoother paths that, if performed by a human, would require fewer button presses. Intuitively one would think this modification would perform identically to the previous NOM; however we have included both for experimental purposes.

3.3.3 Consecutive Action Modifier (CAM)

The consecutive action modifier is a variation of the action change modifier that considers the time between the current action and the previous action. It is therefore defined by two parameters as opposed to previous modifications: the first being the number of frames since the last action (the window), the second being the multiplier. It uses a simple binary check, imposing the penalty on consecutive actions until the threshold has been passed at which point it no longer applies. This results in penalizing multiple actions in quick succession, which we believe leads to more human-like play.

3.3.4 Progressive Consecutive Action Modifier (PCAM)

The progressive consecutive action modifier is the same as the consecutive action penalty: however, instead of being a binary threshold, the value of the multiplier is calculated as a linear interpolation over the window's duration. Given a multiplier M, a frame window F, and considering the number of frames since the last action was performed as F_l we find our interpolated value M_i as follows:

$$M_i = M - \left(\frac{F_l}{F} \times M \right)$$

3.3.5 AVM Integration With the A* Algorithm

Our implementation of the A* algorithm uses the standard formula for node selection from the priority queue based on $f(n) = g(n) + h(n)$, with the next node n to be expanded by the search being the one with the lowest value of $f(n)$. The value of $g(n)$ denotes the sum of the action costs so far to node n in the search (the total path cost), with the A* algorithm attempting to find the path that minimizes $g(n)$. As mentioned previously, due to the nature of the real-time environment we do not use distance as a path cost function, but instead we use game time measured in frames (how much time did it take us to get to the goal, not how far did we travel). This means that the value of $g(n)$ at any time in the search is the current frame count of the game state.

Each of our AVMs act as multipliers on the value calculated by $h(n)$, effectively prioritizing which actions we should select next in the search expansion. Intuitively this modifies the outcome of the search in a way that is similar to Weighted A* Search, where we are willing to reduce the time-optimality of the final path cost as a trade-off for the human-like behavior that we desire. We can also combine the effect of any number of AVMs by simply multiplying them together. For example if we had ACM=0.7 and CAM=1.3, if both applied then our A* search instance would use $f(n) = g(n) + h(n) * 0.7 * 1.3$. The final form of our modifications as they apply to the A* $h(n)$ function are presented in Algorithm 3. Note that if all modifiers are given as 1.0 then the final value of value will remain unchanged from default behaviour.

3.4 Evaluation Metrics

Our evaluation metrics center around the frequency and delay of time required to take actions, which attempts to mimic the action of a human being pressing a button or key. Once an action is selected, such as “move right”, the act of continuing this action continues until the stop action is taken, as a human being holding the button to move right requires no

Algorithm 3: Our final modified heuristic function

```
Input: $N_c$: Child Node, $N_p$: Parent node, $A$ action
Output: $g \in \mathbb{R}$ : g value for A* algorithm
Given:
$N_m$ : No op multiplier $\in \mathbb{R}$
$A_m$ : ACM multiplier $\in \mathbb{R}$
$C_w$ : CAM window length $\in \mathbb{Z}$
$C_m$ : CAM multiplier value $\in \mathbb{R}$
$P$ : Use progressive modifier $\in \mathbb{B}$
begin
    cost = $N_c$.currentFrame - $N_p$.currentFrame
    $T$ = $N_c$.framesSinceLastAction
    if $A$ = "No-Op" then
        cost $* = N_m$
    else
        cost $* = A_m$
    if $T \leq C_w$ & $A \neq$ "No-Op" then
        if $P$ then
            cost $* = C_m + T * \frac{1-C_m}{C_w}$
        else
            cost $* = C_m$
    return cost
```

further input to continue the action. As previously described, the start and end of actions are analagous to humans pressing and releasing buttons on an input device.

Once a path has been calculated through a search instance, we examine the series of actions taken, and our selected metrics are then aggregated to produce a final value representing the model's general performance. In the following subsections, we present the following metrics and their rationales.

3.4.1 Timing Frequency Metrics

Total Actions Performed (TA)

Calculated by summing the number of all actions performed during a search. This is a value we want to minimize and is analogous to how many buttons a human would have to press or release to follow a path created by the search.

Actions per Second (APS)

This metric is calculated by taking the total actions performed and dividing it by the total time spent in engine once the final path has been calculated. The time is calculated in frames, but we present the values in seconds for ease of readability and understanding. Though similar to the previous metric, not all paths are guaranteed to have the same duration. A path may have more actions, but if it is significantly longer, they will occur over a longer period, and may have more time between then. It is analogous to how often, on average, a person would have to press or release a button in order to follow a path created by a search.

Time Between Shortest Actions (TBSA)

While the previous metrics consider all actions taken, this metric uses only the shortest duration between any two actions for a search instance. The shortest required action is of interest because it represents the peak of difficulty for that path. For example: a path might have many long stretches of inactivity followed by a jump that requires two buttons to be pressed within a 5 frame window. The average of such a path would be high, but that would mislead how difficult the path truly is. It is analogous to how fast you have to be able to press two buttons sequentially to be able to a path created by a search.

Given that paths produced by the default A* implementation are often considered superhuman, reducing the average number of actions and increasing the time between any two of them can be considered analogous to creating more human-like pathing.

3.4.2 Robustness and Sensitivity Metrics

Robustness refers to a paths ability to be resilient to the noise which will inevitably be generated by human error in attempting to follow the path. We measure robustness using a form of sensitivity analysis. Once a replay is generated, we perturb the timing of all action changes by shifting them forward or backward by a number of frames. This would correspond to the imperfections in timing that would be observed by a human player. This can also be considered as adding regular noise to the timings of actions required to recreate the path. As an example, humans usually do not wait until the very last frame possible to perform a jump, as being one frame late on their input would cause them to fall through a gap. As such we would say such a path is not as robust as one that would account for more leniency in player input.

A brief overview of our perturbation algorithm can be seen in Algorithm 4. Each search being considered by our experiment generates a replay R which we will use to create a number of perturbed replays, as described above, given as R_n. After the path has been perturbed a number of times, we re-simulate the newly created perturbed replays from the same beginning state, tracking two metrics:

Mean Perturbed Search Displacement Distance (PDD)

Mean perturbed search displacement distance sums up the final position of all instances of the perturbed paths, averaging the distance by which all differ from the original path. It is a measure of the variance of the perturbed paths. A low value indicates that most of the perturbed paths ended up very close to the original endpoint. If, on the other hand, a path gets stuck behind a wall or overshoots a tricky jump it could end up very far from the original endpoint, increasing the mean displacement distance. A lower value means that the path is less sensitive to noise in the input and therefore, we argue, more robust.

Algorithm 4: Perturbing Replays

```
Data: $R$
Result: $RP = [R_1, R_2, ...R_n]$
begin
    $RP \leftarrow []$
    for $i \in [-20, 20]; i \in \mathbb{Z}$ do
        foreach $A$ in $R$ do
            $R_i \leftarrow R$ ;                                  /* copy replay */
            foreach $AP$ in $R_i$ do
                if $AP_t >= A_t$ then
                    $AP_t += i$
            RP.insert($R_i$)
```

Mean Perturbed Deaths per Failstate (PDF)

Mean perturbed deaths per failstate tracks the absolute number of perturbed instances that reach a failure state. This metric is only tracked when any instance of a perturbed replay reaches a failstate, and is ignored otherwise. Because our experiments only include falling into pits as a fail condition, we use "deaths" as a shorthand to avoid the more cumbersome "perturbed failstates per failstate"; however the possibility exists for future non-death failstates to occur for future development. As before, it is a measure of variance, but for the specific case of failure. Failure is particularly important to the perception of players, and repeated failure usually results in frustration especially as many games use a system of lives, or a punishment of time as a consequence of failure. We propose that paths with a lower number of deaths per failstate are more robust, as with PDD, whereas perturbed paths with very high numbers of deaths potentially represent a path that could be frustrating for players to attempt to follow with human reaction times.

These metrics are of particular importance to us, because they objectively test the intuition behind our modifiers. While our modifiers focus on reducing the timing of actions, robustness of a path is not directly targeted for optimization. As such, if any correlation is found it will not be a result of directly targeting this metric, but rather emergent behaviour. Put simply, it would show that paths that require more human-like inputs would generally be safer for humans to attempt to follow.

Chapter 4

Preliminary Analysis

This section provides an more in depth overview of the rationale behind selecting the various AVMs, as well as some screenshots from our engine in order to provide an intuitive sense of the various modifiers. In particular, in order to test real valued parameters we need to decide on a useful subset of discrete values. To accomplish this we would have to quickly test many possible values in order to determine a rough approximation of their lower and upper bounds. A test suite was constructed using our engine with the ability to debug information both with data and visual analysis.

4.1 Realtime Analysis

Within the engine we have the ability to select a destination point and initiate two different searches. The most useful case for this is to test various parameters of our AVMs as compared to default behaviour of A*; however, any two arbitrary searches can be compared using this method. As stated previously, the searches act within the engine in realtime, using only time as a heuristic. Once the searches are performed the two paths are drawn on the screen. Though arbitrary colors may be selected, for the sake of clarity, the first search will be presented in black and will be given as unmodified A* search behaviour. The second path, incorporating AVMs, will be presented in white. An example is presented in Figure 4.1

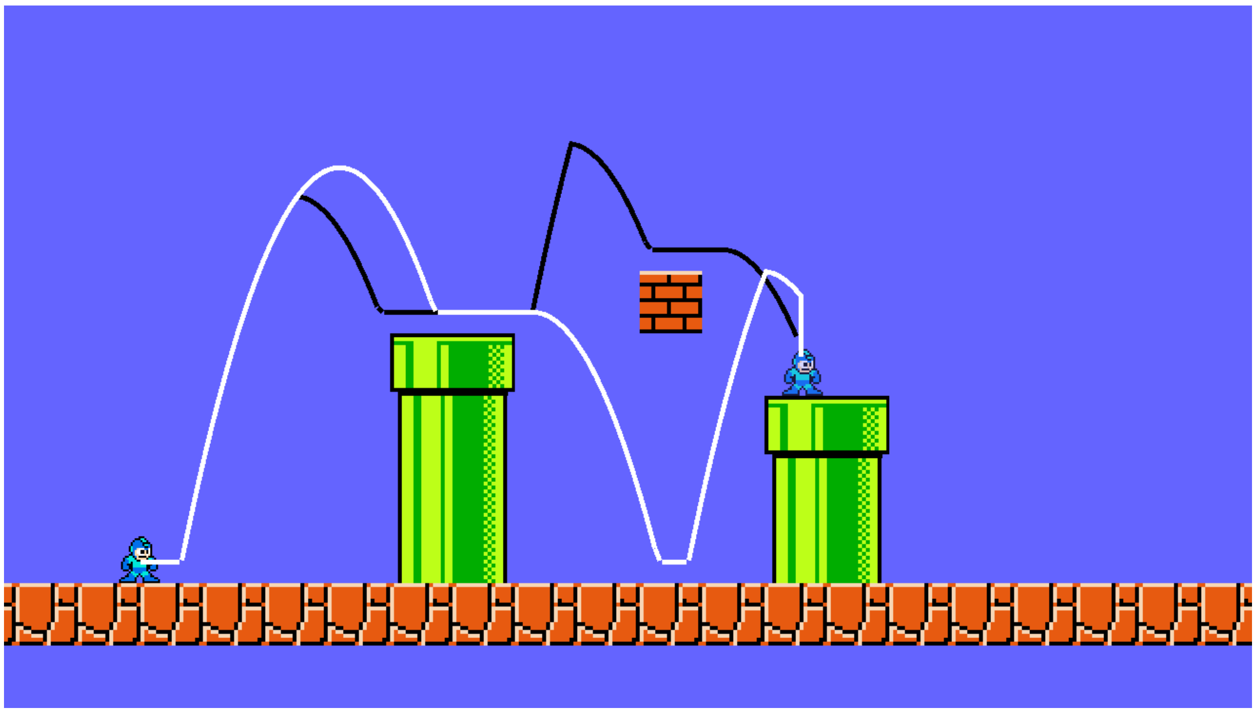


Figure 4.1: An example of how we visually compare differing paths. Default behaviour in black, modified in white.

```
Default: Black                          Modified: White
Last Frame: 156                         Last Frame: 156

Distance: 1669                          Distance: 1495

Displacement: 785                       Displacement: 785

Action Changes: 7                       Action Changes: 4

Shortest Time Between Actions: 12       Shortest Time Between Actions: 18

Longest Time Between Actions: 36        Longest Time Between Actions: 54

Mean Action Change Time: 22.2857        Mean Action Change Time: 39

Search Nodes Expanded: 363              Search Nodes Expanded: 63

Time Elapsed: 165ms                     Time Elapsed: 37ms

End Status: completed                   End Status: completed

Survival Rate: 0.543333                 Survival Rate: 0.659091

Position Deviation: 185                 Position Deviation: 157
```

Figure 4.2: Sample data output from comparing searches in realtime

In addition to visual analysis, we are given an immediate data analysis of the two paths using our previously discussed evaluation metrics. Having this information available in realtime proved extremely useful in preliminary testing, and in combination with the visual output allows for a simple way to quickly attain an intuition of our various AVMs strengths and weaknesses. A verbatim example of our engine's console output is presented in Figure 4.2.

4.2 Analyzing Parameters of Individual AVMs

What follows is a brief description of the behaviours observed through investigating parameters of AVMs. Screenshots of notable behaviour will be provided but as the AVMs will be directly compared through experimentation, detailed analysis may be omitted.

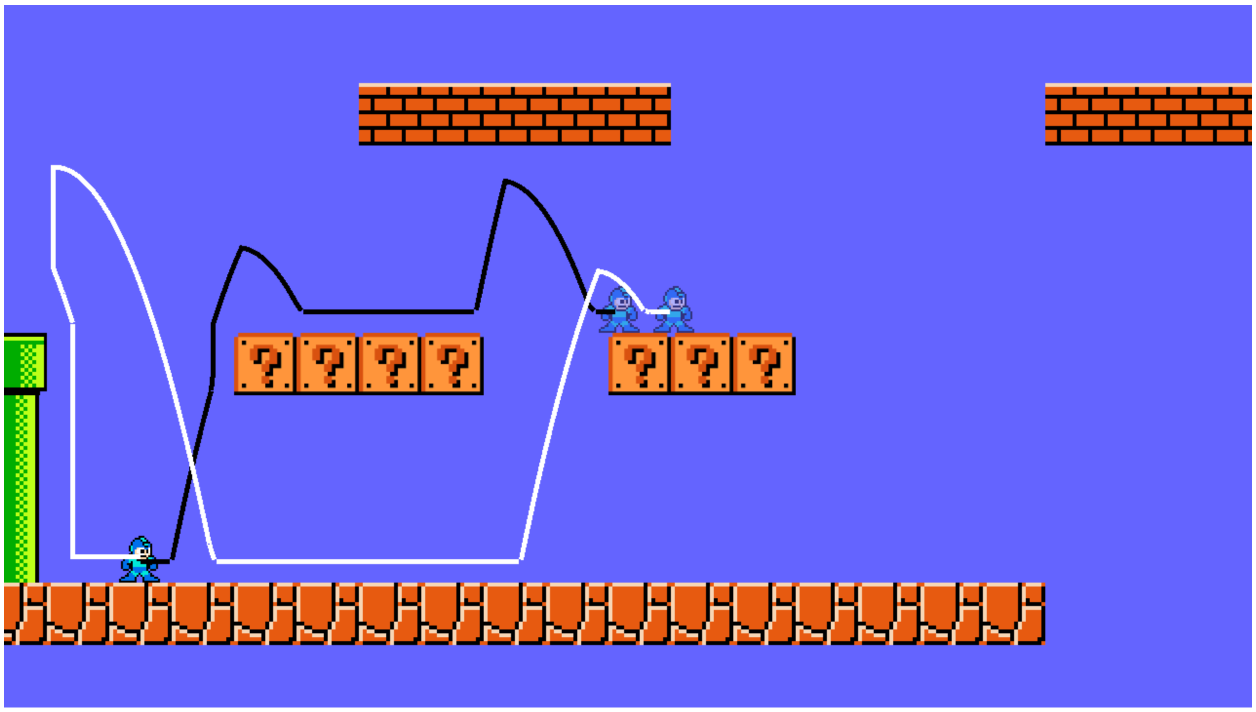


Figure 4.3: The results of using a high number of skip frames, shown in the white path.

4.2.1 Skip Frames

We found that high values of skip frames provided no benefit to making paths safer, simpler, or more humanlike. The obvious pitfall of such a method is the high level of discretization of movement. By advancing the player by such a large degree every simulation step, the algorithm runs into situations where they are unable to reach the range where action is properly required to make a jump. In such cases we have observed the search making odd or nonsensical movements in order to "realign" itself. As an example, in Figure 4.3 we set the white path to use 16 skip frames. At our games framerate of 60 frames per second this corresponds to approximately 260 milliseconds, which is generally understood to be an average human reaction time. The results were unoptimal enough to disregard further experimentation on parameterizing skip frames. A value of 6 was used for the remainder of the experiments.

4.2.2 No-Op Modifier (NOM)

We hypothesized that incentivizing taking no action would result in more human-like behaviour, and visually our suspicious were confirmed, as the NOM provides a noticeably distinct difference from default behaviour. The paths are generally longer, jumps are held for longer, and overall jitter seems to be reduced. Differences become notable at values as low as 0.9, thanks mostly to the default algorithms indifference to jumping frequently (as in our engine jumping does not change the players horizontal speed). This has the benefit of immediately reducing the total number of action changes. This behaviour continues into lower values; however, at values less than 0.1 it became clear we were reaching the limit of the NOMs practical use, as it began to create more and more circuitous paths, valuing sheer length of uninterrupted time more than reaching the goal sensibly (see Figure 4.4). We elected to select values 0.75, 0.5, 0.25 and 0.1 for testing.

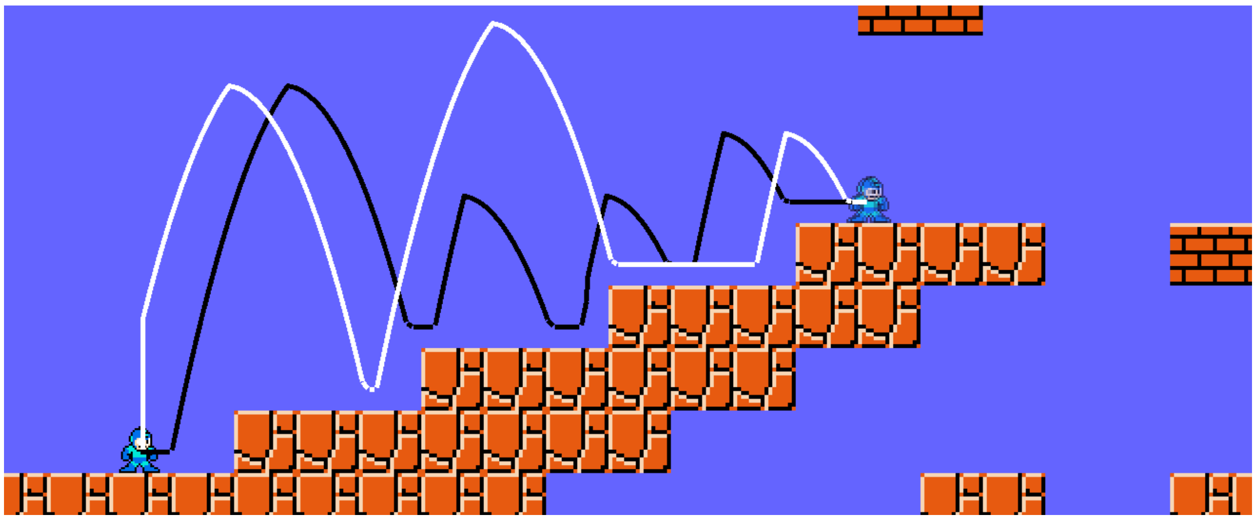
Of particular note is how well the NOM performs as an optimization for time. By essentially acting as a greedy search, this parameter, even at modest values, can decrease the number of nodes expanded by the A* search and consequently the calculation time by an order of magnitude. It should also be noted that as NOP acts as an incentive to a lack of action as opposed to a penalty to taking one, it is the only one of our values whose multiplicative range is effectively $[0, 1]$.

4.2.3 Action Change Modifier (ACM)

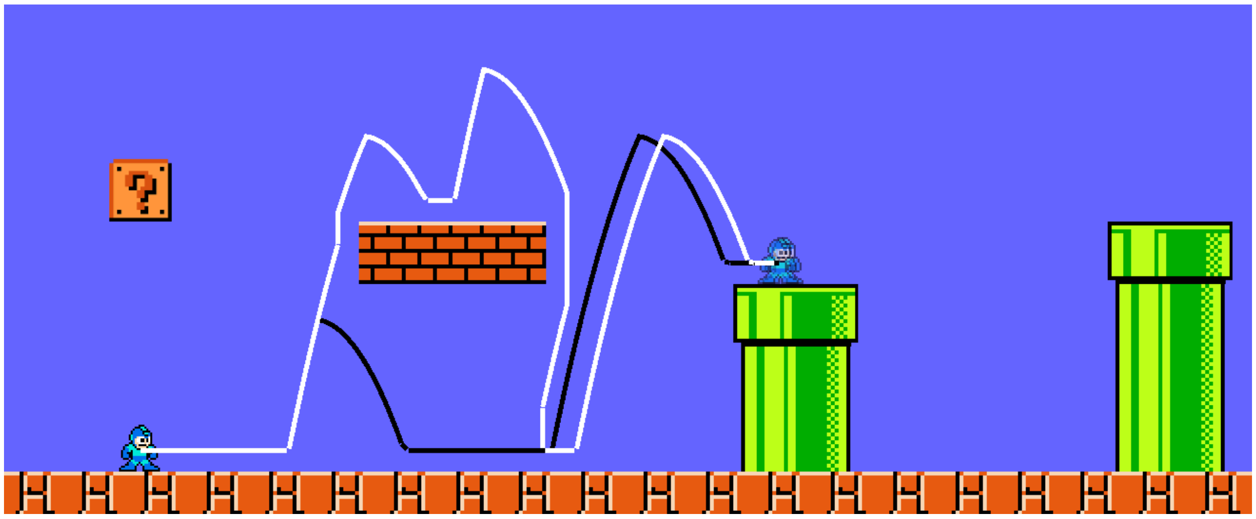
The ACM, being essentially the inverse of the NOP in concept, performs similarly well at values ranging from 2 to 16. It exhibits the same reduction of jittery movement, backed up in data by its ability to reduce the total number of action changes. At higher values it exhibits a better quality of visual performance compared to the NOP as well. However at extreme values because it is penalized so strongly for taking an action, it will often wait until the last possible moment before performing required actions. This can lead it into situations where it avoids danger even less than default behaviour, and indeed values above 16 show a marked decrease in our survival rate metric compared to default, as can be seen in Figure 4.5. Additionally, though it is not our goal to optimize for performance, high values of ACM perform much more poorly than default behaviour. We elected to choose values of 2, 4, 8 and 16 for our experiment.

4.2.4 Consecutive Action Modifiers

The CAM and the similar PCAM both have two variables to select from, and as such provide a much larger space to optimize. However, the frame window within which actions are penalized is not only an integer value, but only has an affect on multiples of the frame skip parameter. As we have selected a frame skip of 6, this made testing somewhat simpler. At frame windows ranging from 6 to 42 they created similarly smooth paths to the previous modifiers, and we found the multiplier can be pushed much higher than the ACM without much degradation in performance in terms of number of actions reduced and sensitivity performance. As can be seen in Figure 4.6, CAM can be pushed to excess, introducing some of the path lengthening we observed with NOP. We elected to test frame windows at values of 6, 12, 18, 24, 30 and 36. Realtime testing revealed no significant difference between the two, and as such would have to be determined through large scale experimentation.

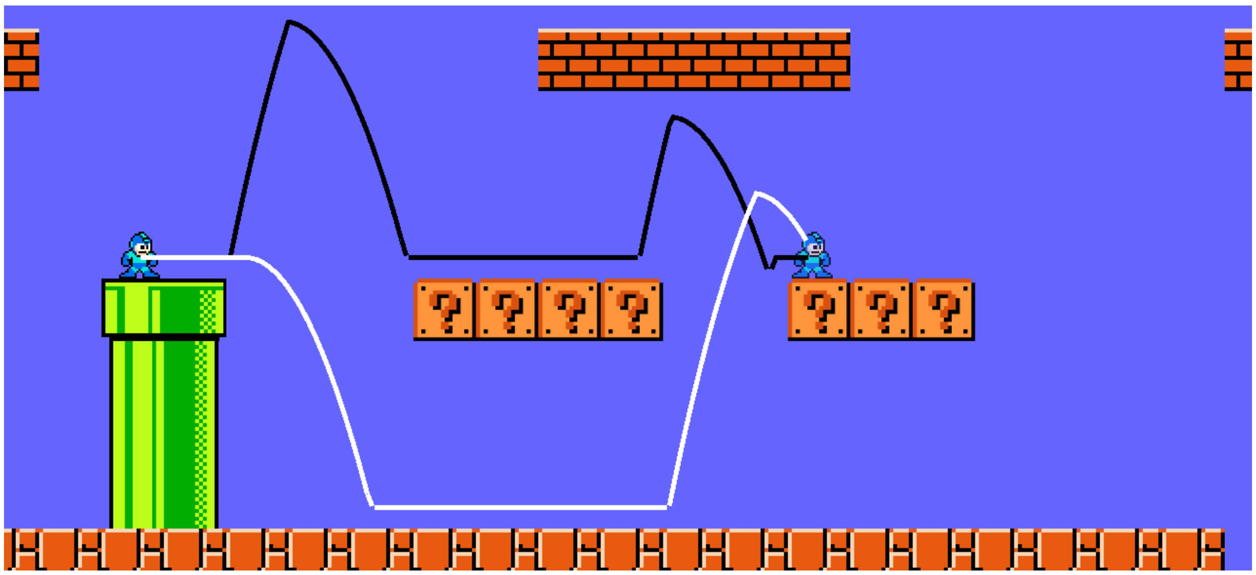


(a) NOM creates longer, smoother paths parameterized at 0.5



(b) NOM can create excessively long and redundant paths parameterized at 0.1

Figure 4.4: The NOM creates excessively long paths at extreme values


(a) ACM performing well parameterized at 4.0

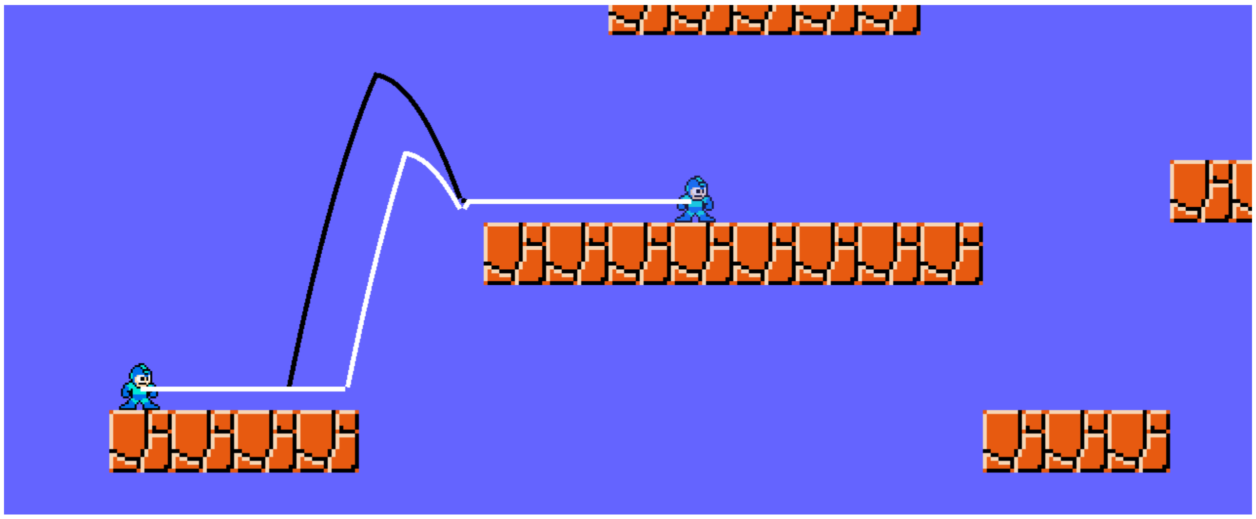
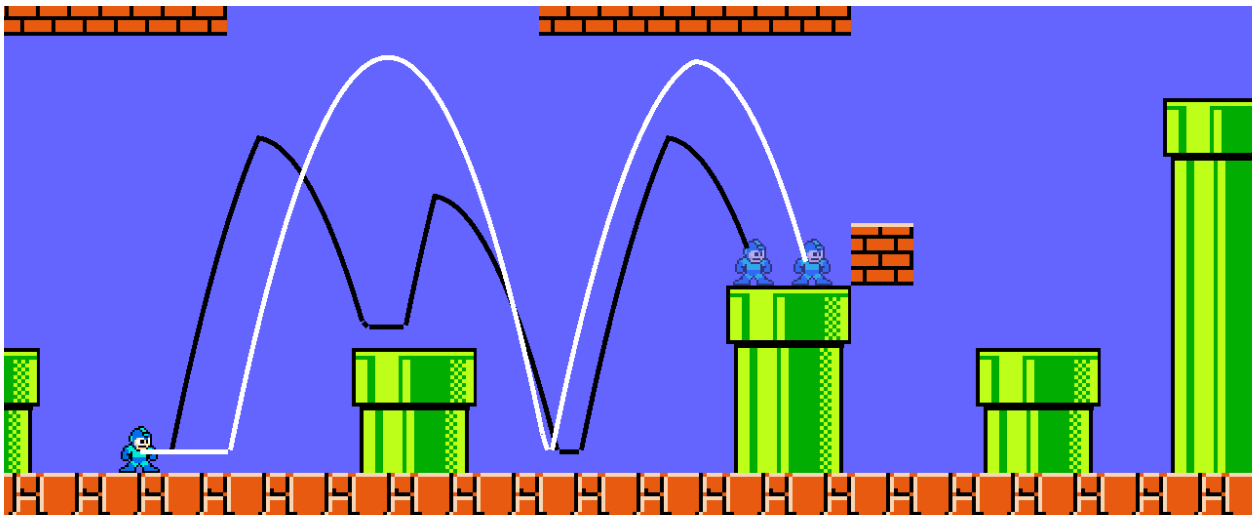
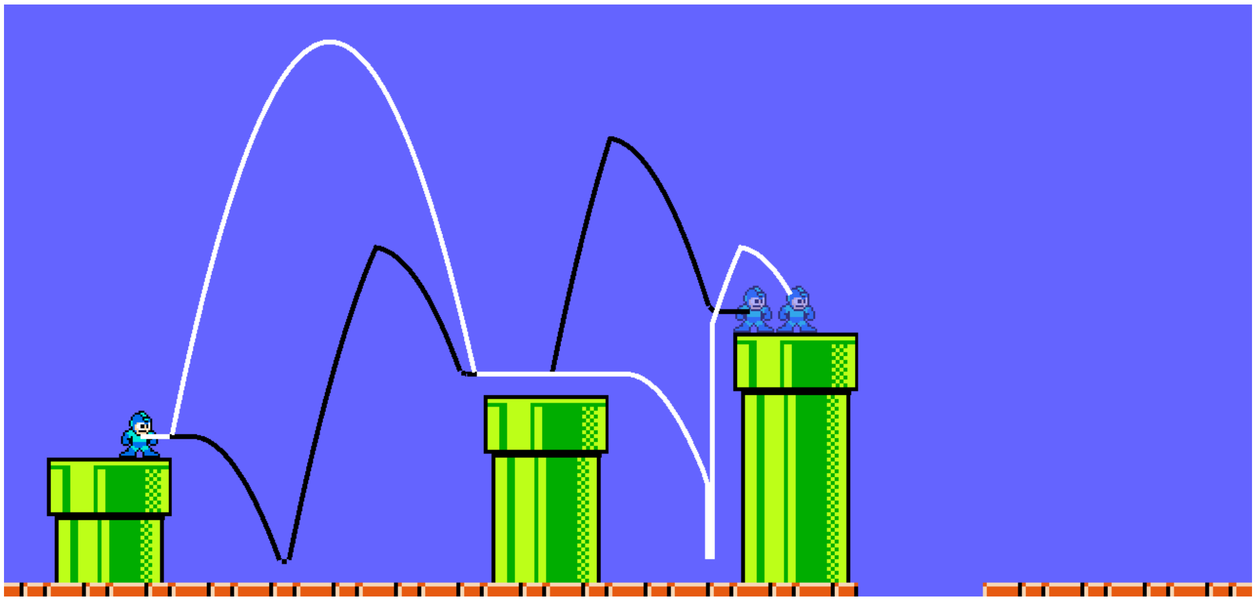

(b) ACM's delaying actions leads it closer to danger parameterized at 32.0

Figure 4.5: The ACM takes riskier behaviour when set too high



(a) PCAM generates smooth paths parameterized with a frame window of 48 and a multiplier of 8



(b) CAM has the potential for excessive length, seen here with a window of 48 and a multiplier of 32

Figure 4.6: Both CAMs demonstrate strong performance even at large multiplier values

Chapter 5

Experiments

This section details the process we used to evaluate a search using our engine, as well as how we tested many instances of searches, modified or unmodified, against each other. Given that each of our AVMs are parameterizable within a range of real numbers, we first sought to find a suitable choice of parameters for each individual AVM in order for it to best represent its optimal performance. Once this was determined, the best performing versions of each modification were tested against each other, as well as a combined search using all best-performing parameters.

5.1 Experiment Process

Our engine uses configuration files loaded at runtime which allows for quick comparisons of many types of searches without the need for recompilation. Each search is defined through variables, the format of which is roughly given in Figure 5.1 (syntax simplified for clarity).

With an arbitrary number of searches defined, we then run a series of experiments. An experiment is simply defined as a collection of searches, as well as some metadata such as the number of iterations to perform, the maximum time allowed for a search, and which search is defined as the baseline (later used to visualize comparisons).

With the experiment defined and loaded by our engine, it proceeds as follows:

```
search params.txt

Name: default

Skip Frames: 6

NOM: 1.0

ACM: 1.0

CAM Window: 32

CAM Multiplier: 1.0

Use PCAM: False
```

Figure 5.1: Example configuration of a search to be used for our experiments.

1. Define several custom search instances through various parameters of our modifiers.
2. Randomly select a level from our level collection
3. Randomly select two walkable tiles a sufficient distance from each other, designating one as the start point and one as the goal point
4. Place the player character at the start point and create a copy of the game state at this point for each instance of search
5. Run each instance of the predefined searches, generating a replay for each
6. Run analysis on each replay file, including the perturbation analysis, saving the data
7. Repeat steps 2-6 for the defined number of iterations
8. Write the data to disk for further analysis

By running each path with every chosen instance of the algorithm, we can be sure that any deviation between the searches is not the result of random selection. Once all iterations are finished, every data point for each individual search is written to a .csv file which is parsed and visualized using python with its packages Pandas[1] and Matplotlib[2].

[1]https://pypi.org/project/pandas/

[2]https://matplotlib.org/

One issue is that of failure. In cases where the chosen path is impossible, all instances of search will timeout and can be ignored. However in instances where one search fails and others succeed we have elected to disregard all searches for that chosen path, as tracking data for different numbers of searches could potentially influence the usefulness of other values. The success rate of searches is still a metric we track, and will be examined later.

Additionally, we found numerous paths existed with only a single action for the entire duration. This is likely an occurrence where the randomly selected start and end points occur on an uninterrupted plane, and as such every search immediately found the best path by holding the corresponding direction button. For the sake of better comparing the unique behaviour of the AVMs, these results were removed from the dataset before any analysis and visualization.

5.2 Selecting Best Performing Parameters

Having decided on a range of acceptable parameters through visual analysis, we ran an experiment in which all the chosen parameters of an individual AVMs were compared against default behaviour. Since multiple metrics were being tracked per instance, we needed to assign some kind of evaluation metric to determine the best performing parameter. Our approach was to take all of the metrics defined in Section 3.4 and evaluate each based on a normalized score, where all values were normalized to a range of $[0, 1]$ within the column, then summed to give a normalized sum which was then divided by the number of metrics to give us a normalized score. In this system a score of 1 means the search gave the worst performance in every category, whereas a 0 would mean the search had the best performance by every single metric.

The Time Between Shortest Actions (TBSA) metric is unique in that we want it to increase, and so to conform to the other metrics all values were made negative before normalization.

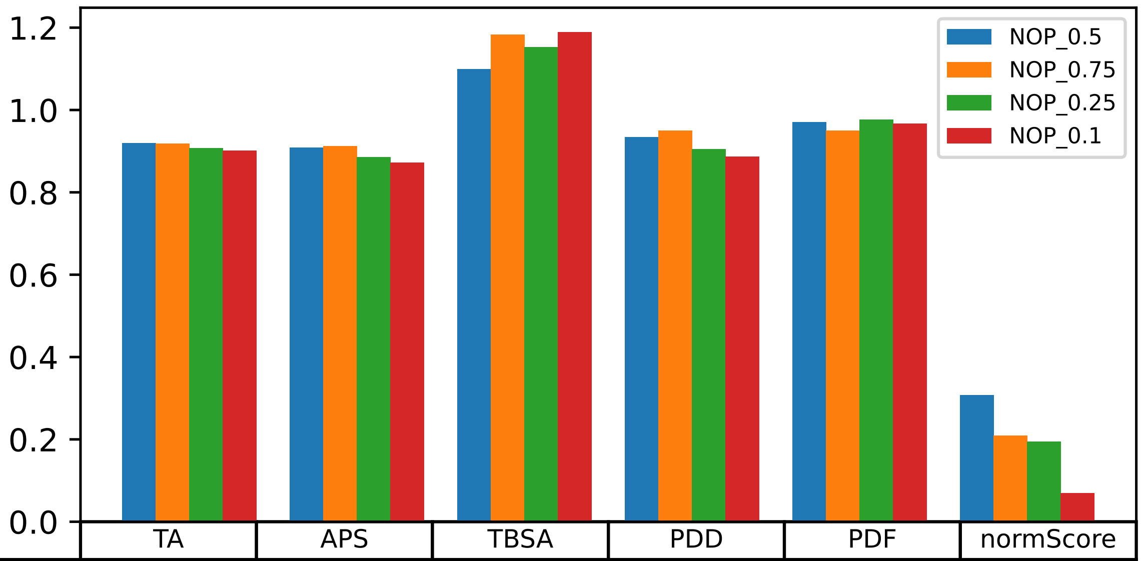
For visualization in the following figures, all values are given as their ratio relative to default, as the specific values such as total number of actions is arbitrary and dependent on the randomized search points as well as the number of iterations chosen for the experiment. As previously mentioned, the TBSA metric is intended to increase above default, and as such values in excess of 1.0 are to be expected, the higher the better, as they represent a relative lengthening of the shortest actions performed by the search. For the CAM and PCAM searches, 18 different parameterizations were selected and for brevity only the top 5 values will be displayed. For the CAM and PCAM, values will be displayed in a hyphenated format where the first value represents the window and the second value the multiplier. Therefore "PCAM 24-6" will mean a window of 24 frames with a multiplier of 6.0 to the path cost.

5.2.1 NOM Parameter Evaluation

The No-Op Modifier showed a general trend downwards (see Figure 5.2). Surprisingly however, a parameterization of 0.75 scored better than 0.5, thanks entirely to the TBSA and PDF being unusually low compared to both 0.5 and 0.25. The value of 0.1 scored the highest in our tests, which was somewhat surprising after visual inspection, as it was selected to be the lower end of what was thought feasible. The best performing metric in this category was the APS score for the 0.25 parameter, which performed at 88.6% of default; however most values seem to be performing in the mid 90% range.

5.2.2 ACM Parameter Evaluation

The Action Change Multiplier performed much more in line with our initial analysis (see Figure 5.3. The value of 16 was selected as what we believed to be the upper end of viability, and indeed it performed worse than the next lowest value of 8. Once again however we noted that the lowest parameter of 2 scored slightly better than 4, thanks entirely to having the lowest PDD score. When un-normalized however, the results were less impressive with a score of only 95.8% compared to the the next best performing search in that metric scoring 94.8%. Despite this, the value of 8 remained a clear winner, and was selected as the best



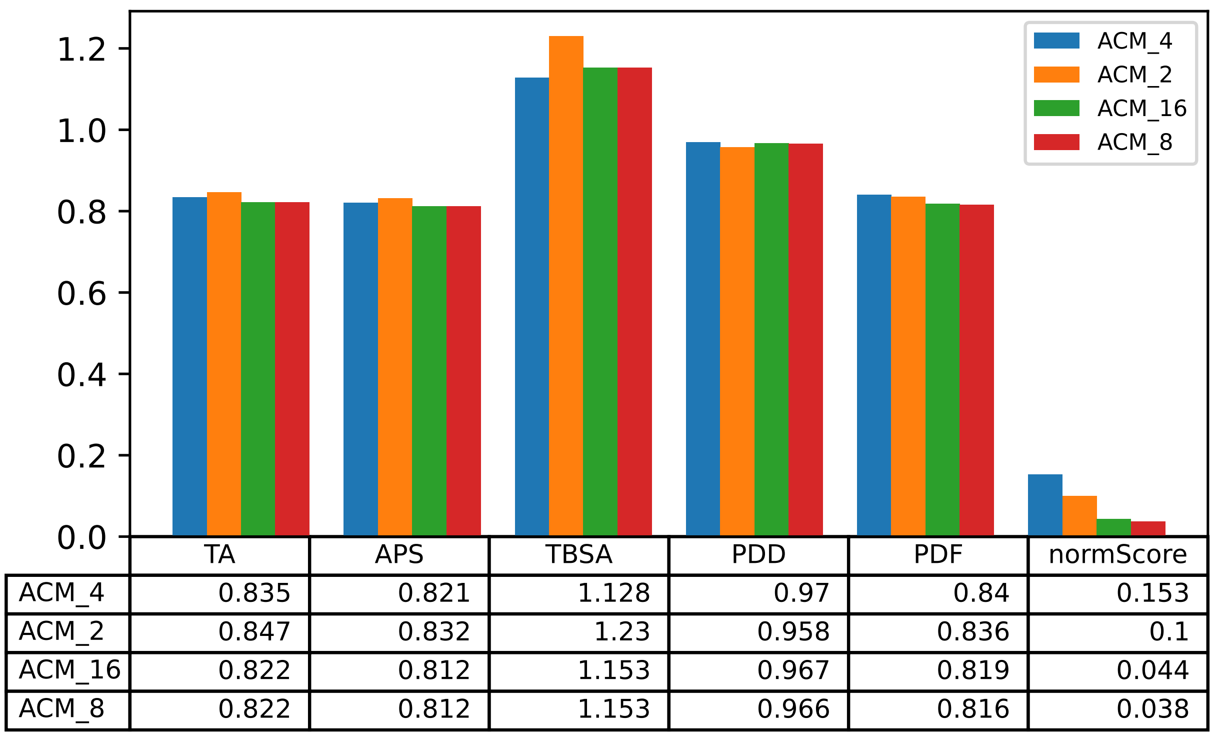
	TA	APS	TBSA	PDD	PDF	normScore
NOP_0.5	0.92	0.909	1.1	0.934	0.971	0.308
NOP_0.75	0.919	0.912	1.183	0.95	0.95	0.21
NOP_0.25	0.908	0.886	1.153	0.905	0.977	0.195
NOP_0.1	0.901	0.872	1.189	0.887	0.967	0.07

Figure 5.2: NOM performance relative to default

performing parameter value. On average the NOM seems to perform better than the NOP with some scores in the low 80% range.

5.2.3 CAM Parameter Evaluation

The Consecutive Action Modifier was able to achieve more favorable ratios relative to default than the previous AVMs (see Figure 5.4). There was a strong general trend towards the higher values being scored lower, with an even clearer bias toward higher multipliers when compared to large window values. Of particular note is the PDF score of higher values, which was able to achieve a low of 41% relative to normal, as well as the CAP 36-8 achieving a PDD score of 72% relative to normal.

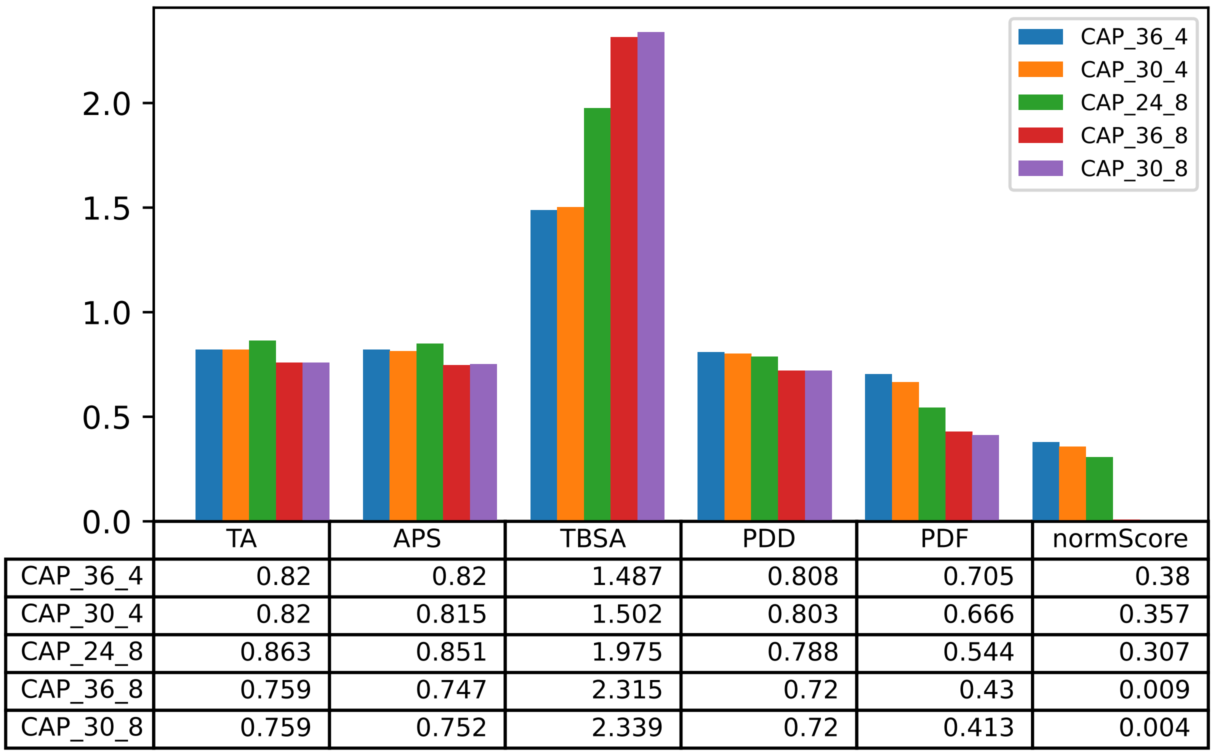


	TA	APS	TBSA	PDD	PDF	normScore
ACM_4	0.835	0.821	1.128	0.97	0.84	0.153
ACM_2	0.847	0.832	1.23	0.958	0.836	0.1
ACM_16	0.822	0.812	1.153	0.967	0.819	0.044
ACM_8	0.822	0.812	1.153	0.966	0.816	0.038

Figure 5.3: ACM performance relative to default

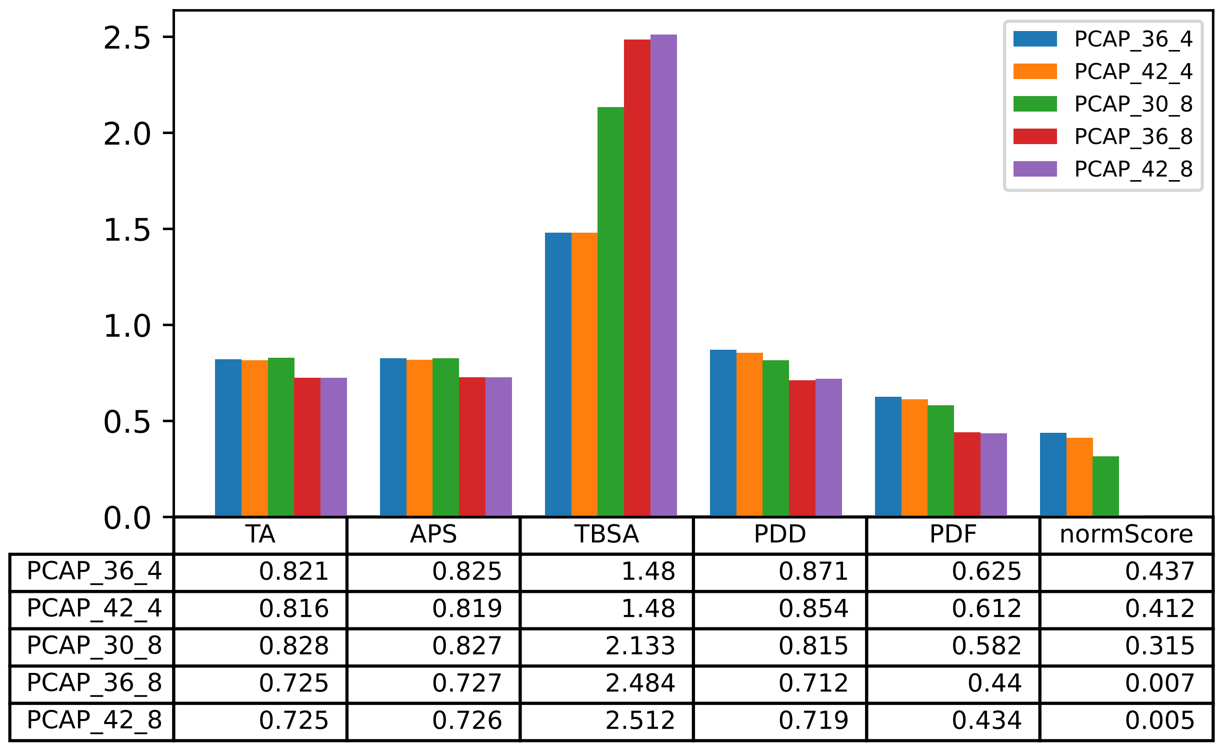
5.2.4 PCAM Parameter Evaluation

Like the CAM, the Progressive Consecutive Action Modifier achieved better ratios over default than the AVMs that did not consider time between actions (see Figure 5.5). As before, there was a general trend towards the higher values performing better, though not in a strict ordering. For example, the 24-4 parameter though not being the highest value for either windowing or multiplier, was among the top 5 performing searches. The best performing, however, was indeed the modifier with the highest value in both categories: a 42 frame window with a multiplier of 8. It performed extremely well in all categories, generally outperforming the best CAM model in every metric except PDF where it scored a 43.4% compared to the 41.3% achieved by the CAM. High scoring searches in the CAM category are achieving 70% of default scores on average.



	TA	APS	TBSA	PDD	PDF	normScore
CAP_36_4	0.82	0.82	1.487	0.808	0.705	0.38
CAP_30_4	0.82	0.815	1.502	0.803	0.666	0.357
CAP_24_8	0.863	0.851	1.975	0.788	0.544	0.307
CAP_36_8	0.759	0.747	2.315	0.72	0.43	0.009
CAP_30_8	0.759	0.752	2.339	0.72	0.413	0.004

Figure 5.4: CAM performance relative to default

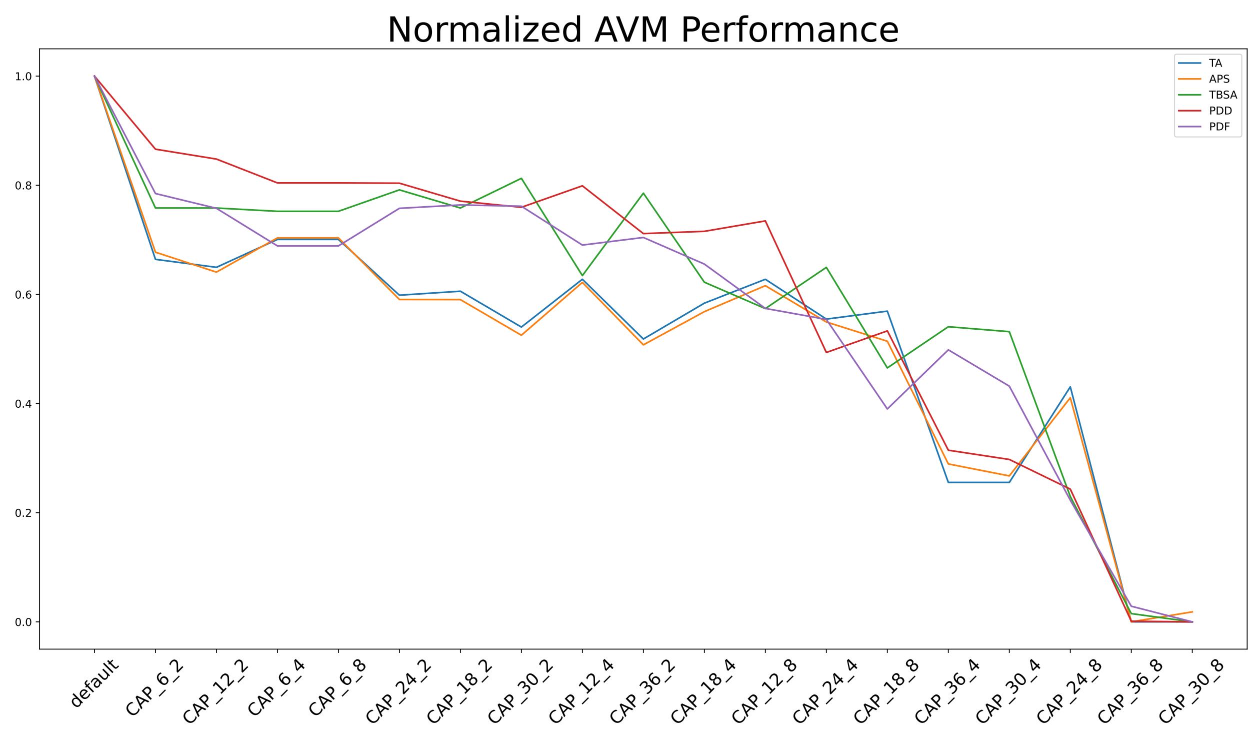


	TA	APS	TBSA	PDD	PDF	normScore
PCAP_36_4	0.821	0.825	1.48	0.871	0.625	0.437
PCAP_42_4	0.816	0.819	1.48	0.854	0.612	0.412
PCAP_30_8	0.828	0.827	2.133	0.815	0.582	0.315
PCAP_36_8	0.725	0.727	2.484	0.712	0.44	0.007
PCAP_42_8	0.725	0.726	2.512	0.719	0.434	0.005

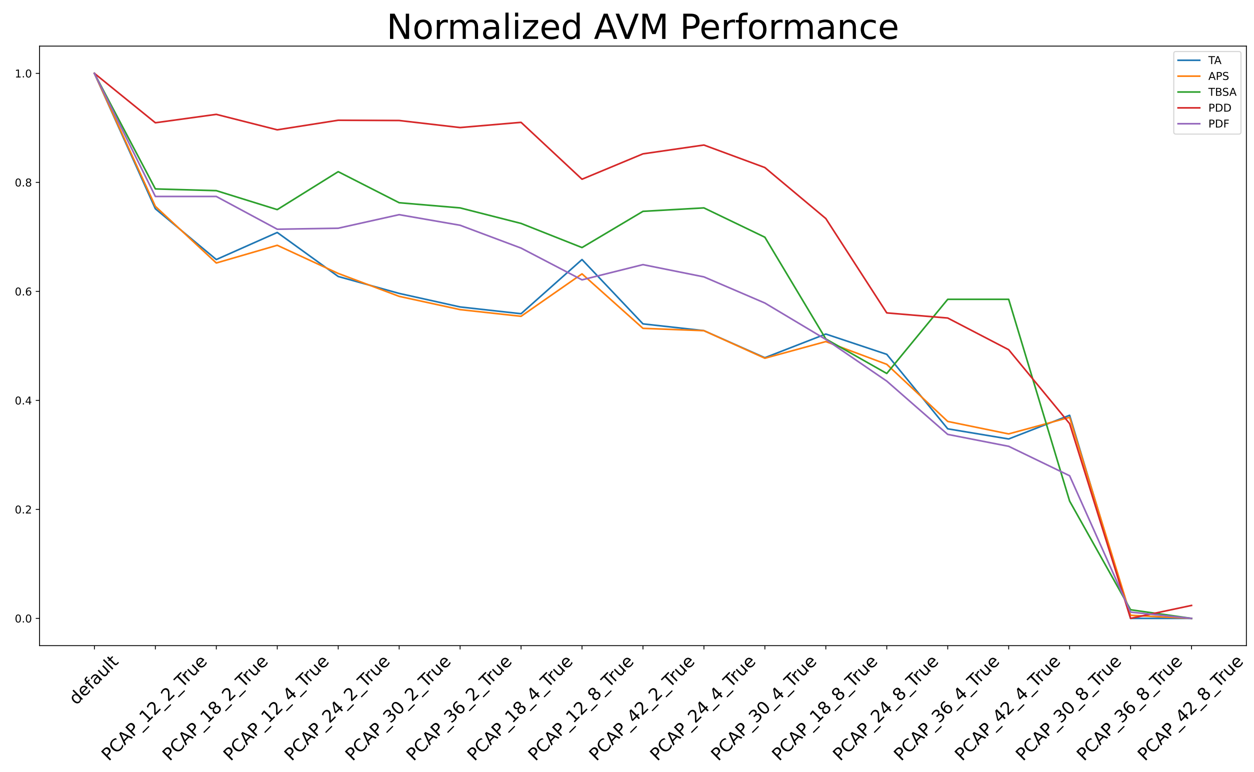
Figure 5.5: PCAM performance relative to default

5.2.5 Linearity of Normalized Scoring

Given that some of the results presented may seem unintuitive in terms of ranking the models, we have constructed a visualization of the search performance ranked by our normalization score in order to show the trends. Figure 5.6 shows all such parameterizations, displayed in order of our normalization score. Each is given in its normalized form, where 1 represents the worst performing model and 0 represents the best. A downward trend from left to right across all metrics is evident.



(a) CAM parameters, left to right by normalized score



(b) PCAM parameters, left to right by normalized score

Figure 5.6: Visualizing the parameters in order of our normalized scoring system.

5.3 Comparing Best Performing Parameters

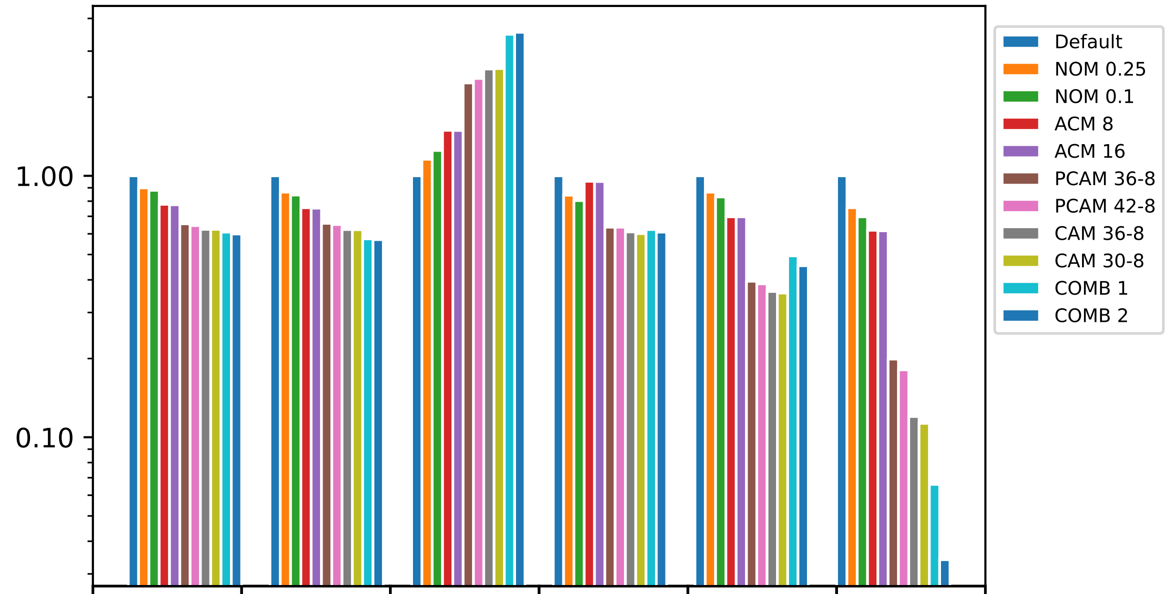
With our input space significantly reduced, we elected to select several of the best performing AVMs and compare them directly against each other. The experiment would be performed as before; however instead of comparing each AVM against itself, they would all be directly compared on the same large number of randomized searches. With many fewer searches in this experiment, we elected to run 2000 iterations to further reduce noise and variance. Additionally we elected to run some combinations of the best performing parameters, to see if using multiple AVMs in combination would give better results than the searches being run on their own. As all the AVMs are multiplicative, their values can be combined freely without regard for order.

We have selected the two best performing parameters from each search as well as several combined values: NOM values of 0.25 and 0.1, ACM values of 16 and 8, CAM values of 36-8 and 30-8, PCAM values of 36-8 and 42-8, and finally two combination searches. Combination search 1 (COMB1) combines all the best performing parameters. It uses PCAM with a window of 42 and a multiplier of 8, an ACM of 8 and a NOM of 0.1. Combination search 2 (COMB2) uses CAM 36-8, ACM of 4 and a NOM of 0.5 in an effort to reduce the influence of the less impactful parameters.

5.3.1 Overview of Results

Our choice of metrics as they apply to this experiment are given in Figure 5.7. Unsurprisingly, the NOM and ACM searches fared the worst. Though all improved upon default search behaviour, none managed to best any of our CAP or PCAP searches in any metric. Of these simpler metrics, the best performing, ACM 16, scored 0.644 using our normalized scoring system, losing to the worst performing consecutive action based model, the PCAM 36-8, with a score of 0.194.

What did come as a surprise on the other hand, was the performance of the CAM compared to its linear cousin the PCAM. The standard CAM performed better in all regards,



	TA	APS	TBSA	PDD	PDF	normScore
Default	1.0	1.0	1.0	1.0	1.0	1.0
NOM 0.25	0.899	0.866	1.157	0.843	0.866	0.755
NOM 0.1	0.88	0.844	1.25	0.804	0.83	0.697
ACM 8	0.778	0.755	1.493	0.953	0.697	0.619
ACM 16	0.775	0.752	1.491	0.951	0.697	0.616
PCAM 36-8	0.655	0.658	2.266	0.636	0.395	0.199
PCAM 42-8	0.645	0.651	2.357	0.636	0.386	0.181
CAM 36-8	0.624	0.623	2.562	0.61	0.361	0.12
CAM 30-8	0.624	0.622	2.572	0.601	0.356	0.113
COMB 1	0.609	0.574	3.478	0.623	0.494	0.066
COMB 2	0.599	0.57	3.54	0.609	0.453	0.034

Figure 5.7: The results of comparing all AVMs using our selected metrics. Values are presented relative to default, and graphed on a logarithmic scale.

perhaps suggesting that higher multiplier values would fare better for the PCAM, or simply that a tighter falloff for the penalty time window used in the CAM is better at reaching our targeted metrics.

Interestingly, the two combination searches performed the best in nearly every metric. The overall winner was COMB 2, which used the standard CAM as opposed to the progressive version, and uses less intense values of the other AVMs compared to COMB 1. Whether the stronger showing is related to the performance of CAM vs PCAM or the lessened value of the other AVMs was unclear.

5.3.2 Action Frequency Metrics

The Total Action metric is the metric most strongly correlated with our normalized score, and as such when sorted by this metric we see a strictly linear relation. Stronger parameterization of NOM and ACM increase the value, and the PCAM with the larger window reduces the total number of actions as well. The CAM performs better than the PCAM, as noted earlier; however, both the 36 and 30 frame window performed the exact same number of actions, perhaps indicating we have reached the upper bound of usefulness for a window at 30 frames. The best performing searches in this category were the two combined AVM searches, with the strongest, COMB 2 managing to achieve 59.9% of the number of actions made by the default algorithm.

	TA	APS	TBSA	Score
TA	1.000	0.993	0.901	0.984
APS	0.993	1.000	0.915	0.977
TBSA	0.901	0.915	1.000	0.932
Score	0.984	0.977	0.932	1.000

Figure 5.8: Correlation matrix of action frequency metrics and our normalized score

Actions per Second is a strongly associated metric with Total Actions. The only instance in which they should differ significantly is in cases where the total time spent playing differs significantly. An excessive discrepancy could be associated with excessively long paths, as observed in Chapter 4. Fortunately, total travel time was a metric we recorded in our searches, given in Table 5.1. It was not included as a key metric as even preliminary testing showed very little variation in time with appropriate values. The worst performing search in terms of travel time in our test, COMB 1, showed less than a 2% increase in total path time. Achieving 57.4% of the actions per second with such a small increase in total time is, we argue, a relatively clear indication of optimization of movement as opposed to a simple lengthening of the path.

Search	Total Time (s)	Ratio Relative to Default
Default	1677.9	1
NOM 0.25	1691.4	1.008
NOM 0.1	1695.8	1.011
ACM 16	1671.9	0.996
ACM 8	1672.1	0.997
CAM 36-8	1678.7	1.000
CAM 30-8	1679.3	1.001
PCAM 36-8	1676.7	0.999
COMB 1	1707.3	1.018
COMB 2	1695.5	1.010

Table 5.1: The total time in seconds taken by all paths of all searches when performed in-engine.

The final metric concerning action frequency is the mean time between shortest actions (TBSA). This is where our AVMs showed the largest improvements over the default algorithm. The NOM and ACM searches performed better than default, improving between

15 and 50 percent; however, every search with a consecutive action multiplier was able to increase the value of this metric by over 200%. The strongest search in this category, again COMB 2, was able to outperform the default by 354%. This was behaviour that we explicitly programmed for, nevertheless; a similar explicit punishment of performing any actions was implied by both the ACM and NOM, which fared much worse.

Explicit Timing

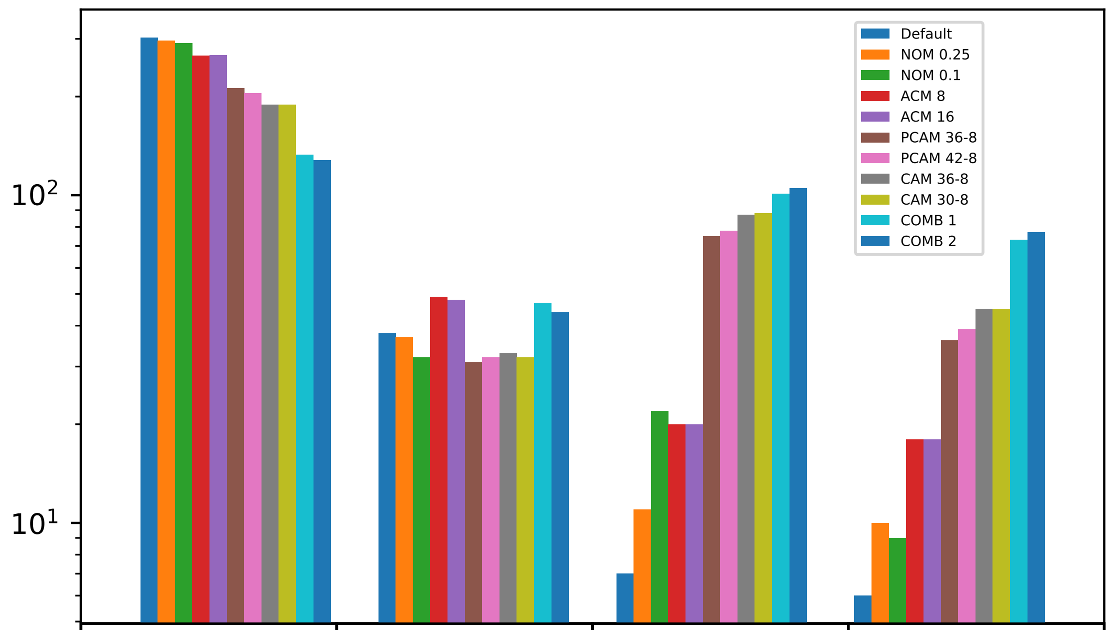
While relative values for timing are useful in assessing relative performance, the absolute values can be useful in comparing against known human benchmarks. Recall that minimum human reaction time is often given as 250ms, corresponding to approximately 16 frames. With our chosen values for the CAM and PCAM windows, the average shortest action required for a path was increased from 10.5 frames at default, considered superhuman, to a more reasonable range of 23-37 depending on the search. This indicates that on average, paths generated with our search technique are indeed human achievable. Interestingly, the 16 frame benchmark of average human reaction time perfectly divides the default, NOM and ACM searches from those using a form of consecutive action penalty.

With these explicit values in mind, we decided to further examine the TBSA metric. An overall average was useful in showing an increase in the minimum required action timing, but we specifically sought to find how many of those actions were considered superhuman. A binning approach was used, where all values of the TBSA for all searches were sorted into 4 bins based roughly on how difficult it would be for a human to accurately time such actions:

- 0-12 frames (0-200ms): Very difficult to perform accurately
- 13-24 frames (200ms-400ms): Difficult but achievable
- 25-60 frames 400ms-1 second: Not difficult
- 60+frames (>1 second): Trivially achievable

The results of this decomposition are given in Figure 5.9. Within each bin the bars are plotted using our normalized scoring system, with default at the left, and the searches continuing

Figure 5.9: TBSA values per search, sorted into bins



	0-12	13-24	24-60	60+
Default	303	38	7	6
NOM 0.25	296	37	11	10
NOM 0.1	291	32	22	9
ACM 8	267	49	20	18
ACM 16	268	48	20	18
PCAM 36-8	212	31	75	36
PCAM 42-8	205	32	78	39
CAM 36-8	189	33	87	45
CAM 30-8	189	32	88	45
COMB 1	133	47	101	73
COMB 2	128	44	105	77

right in order. A clear progression can be seen. In the category deemed superhuman there is a downward slope as the searches progress in scored order, whereas in the higher values this trend is reversed. This indicates that the average did not simply increase for example as a slight increase from 6 frame actions to 12 frame actions, but rather that an overall lengthening occurred across the entire range of action times. In the 60 frame plus bin equating to one second between actions, the default search only performed 6 such actions across the entire experiment whereas COMB 2 (the highest performing search) performed 77.

One final point on absolute timing is the extremely similar performance between the CAM searches parameterized at 30 and 36. As 30 frames corresponds exactly to half a second, this may indicate that our level corpus (Super Mario Bros) does not expect reaction times faster this reaction window. As it is roughly twice the time of what is considered normal human reaction time, this could perhaps be in line with the designers intentions.

5.3.3 Robustness Metrics

Whereas the results of the action based metrics are indeed promising, the robustness metrics represent behaviour not explicitly programmed into our AVMs. By perturbing, ie. adding noise to our generated paths, we sought to generate data on how likely a human player would be to either diverge from the intended route through imperfection, or worse still, find themselves in danger through a simple mistiming. Perturbations ranged from +-20 frames for each action in a search replay, which slightly exceeds the previously established average reaction time. In doing so, we propose to emulate a wide range of human ability in attempting to follow the paths generated by our searches.

	PDD	PDF	Score
PDD	1.000	0.878	0.930
PDF	0.878	1.000	0.958
Score	0.930	0.958	1.000

Figure 5.10: Correlation matrix of robustness metrics and our normalized score

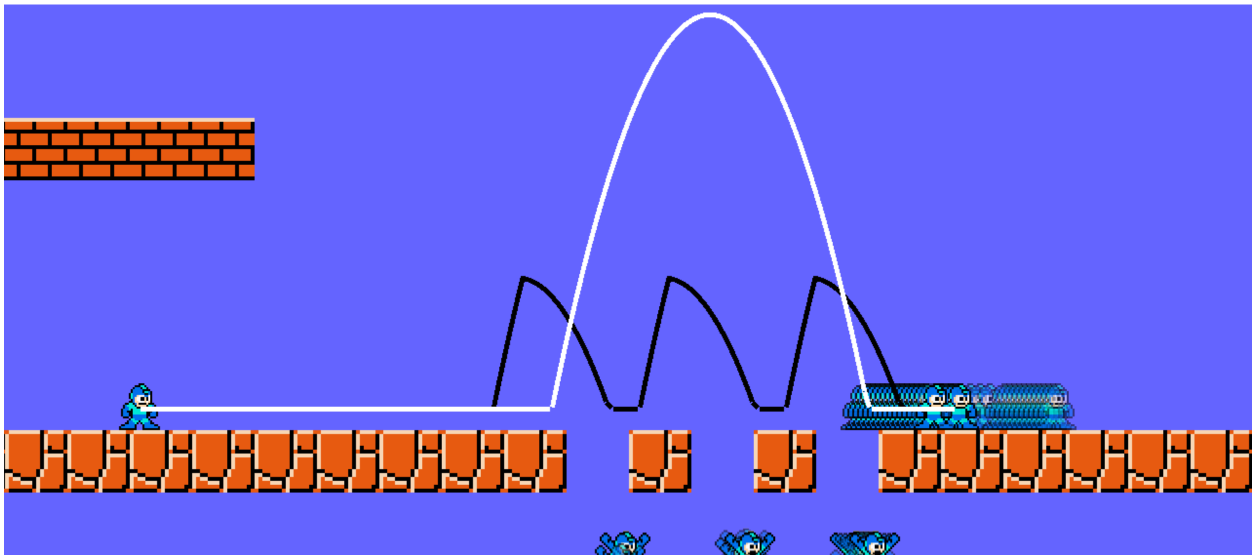
Mean Perturbed Displacement Distance (PDD) sums the final position of all paths and calculates the distance of the final position in pixels. For reference, a single "block" in our engine is 64x64 pixels. The default search algorithm had a PDD of 133.8 pixels across the entire experiment. While this equates to an average of more than two blocks, visual analysis of the algorithm shows this metric to have an extreme amount of variance. Failing to surmount an obstacle will often leave a perturbed path stuck, unable to continue, which greatly adds to the mean distance. Fail states similarly add to the displacement distance, as their position is set at the point of failure.

Once again, every chosen metric reduced the value of the perturbed displacement distance. Of the ACM and NOM values, despite the ACM performing better overall, the NOM showed a marked improvement in perturbed displacement distance, with both NOM searches outperforming both ACM searches. Recall from our preliminary analysis that the NOM tended to extend the path length at high values, whereas the ACM did not exhibit such behaviour.

All of the remaining searches performed very similarly, with less than a 3% difference between the sixth best performing search and the first. A slight difference nevertheless exists between all of the CAM and PCAM based searches, with every search incorporating CAM once again performing better than every PCAM search, including the combined searches.

Our initial impressions were that the Perturbed Deaths per Failstate (PDF) would simply be a subset of displacement distance, where a sufficiently displaced path over a jump would be one sufficient for failure. However our expectations were subverted, as the two share some of the weakest correlations of the major metrics. This is most plainly evidenced by the ACM and NOM searches again. Whereas the NOM performed better in terms of perturbed displacement distance, the ACM performs remarkably better in terms of avoiding failstates.

All subsequent searches, however, faired much better on the PDF metric, to the point where it represents the largest percent reduction compared to default of all metrics. There is a marked jump between ACM 16 and PCAM 36-8. Despite being subsequent in terms of our normalized score, the PCAM search has 56% fewer deaths on average. Once again the standard CAM slightly outperformed the PCAM in this metric, as well. Interestingly this is one instance where the two combined searches fared markedly worse than the single CAM or PCAM searches. Indeed, the robustness metrics are the only metrics in which COMB 2 did not perform the best.


(a) Perturbed routes of the default path (black) show large variance in end points

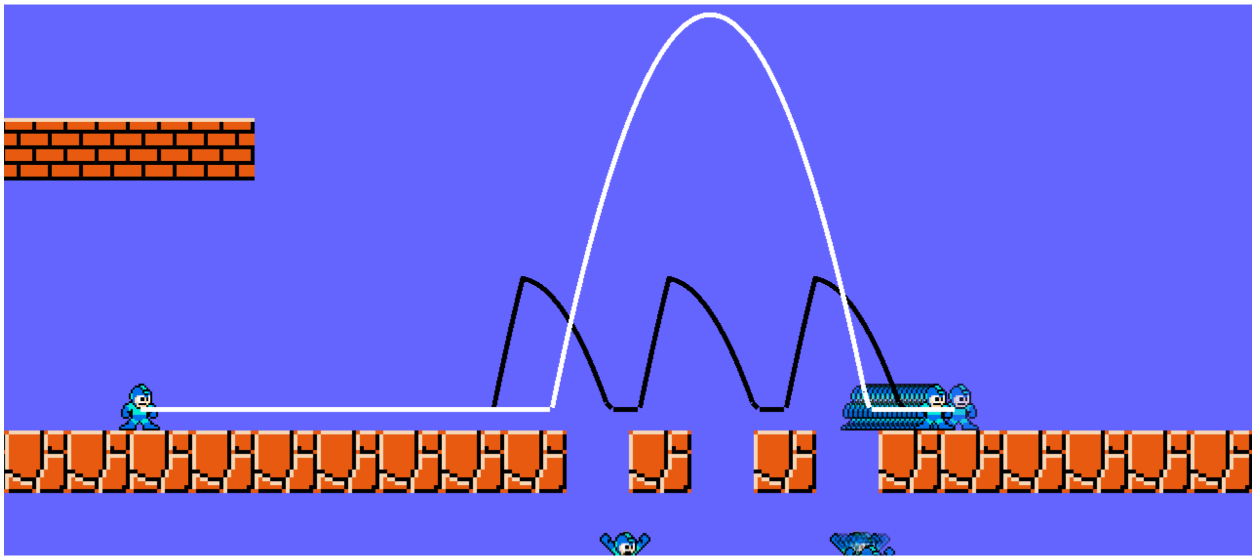

(b) Paths created using the best performing AVMs (white) avoid danger and create less perturbed variance

Figure 5.11: A demonstration of robustness. Perturbed player endpoints are shown as transparent figures. The figures at the base of the pits have reached fail states.

5.3.4 Search Instance Calculation Time

Efficiency of calculation was not an intended goal of our algorithmic modifications to the A* algorithm. Nevertheless, we tracked two metrics related to efficiency: the total runtime of each search instance, as well as the total number of nodes expanded in the search tree. These two metrics combined allow us not only to determine the efficiency of the algorithm in a general sense of how long path calculation takes, but by tracking the total number of nodes expanded, we can demonstrate efficiency in terms of how likely the algorithm is to greedily select the best nodes to meet its own goals. Even with two identical paths, if one algorithm expanded fewer nodes on the way to finding the path, it can be said to be a more efficient heuristic. Our efficiency findings are presented visually in Figure 5.12.

The number of nodes expanded per search is given in Figure 5.12a and indicates the aforementioned greedy-like efficiency of the search. Unsurprisingly default A* fared the worst in this regard, expanding an average of 128.8 nodes per search. The no-op modifiers performed the best, with each expanding approximately 47 nodes per search. The combined searches, likely as a result of their inclusion of no-op modifiers also performed well. Because this metric has the most practical impact on the searches in terms than sheer runtime, all graphs in Figure 5.12 are sorted from worst to best in terms of their nodes expanded.

The mean calculation time of all searches is given in Figure 5.12b. This value does not account for the number of nodes expanded, and as such was not guaranteed to follow a similar trend. The ACM modifier is the standout here, with ACM 16 nearly matching the default, and ACM 8 performing worse than its position in the nodes expanded hierarchy. It would seem that any performance gains from the node selection behaviour of the ACM modifier are offset entirely by the additional time spend calculating the new heuristic values. Once again the NOM performed the best in this category, taking approximately 20ms per search as opposed to the default 58. Both cases equate to an approximate 70% reduction compared to default behaviour, and so unlike the ACM, this reduction in time is likely strictly related to the reduction in nodes expanded.

The final metric tracked is the nodes per milisecond expanded by each search, given in Figure 5.12c. As this value has been normalized per node expanded, it indicates the amount of processing time added by our modifications. Though there is variance it seems randomized and far less significant than the differences between the previous two metrics. Statistically, nodes per milisecond is not strongly correlated with either processing time or nodes expanded. It should be noted as well that noise is likely to be added to these values from the operating system and inconsistent nature of CPUs, and as such we do not believe much can be inferred from this metric apart from the fact that none of our AVMs significantly affect the processing time required per node of the A* algorithm. Any efficiency gains or losses are likely the result of node selection imparted by the modified heuristic.

5.3.5 Additional Metrics

While our selected metrics were deemed most significant in terms of generating human achievable paths, several other metrics were tracked throughout the experiment for the sake of interest. What follows is a brief discussion of these metrics.

Jumps

The core of most platforming games is jumping, and as such we specifically tracked the number of jumps performed by our searches. This metric was excluded from the core set in the interest of generality, but is nevertheless interesting. Figure 5.13 shows the frequency of jumps per search, along with the average jumps throughout the entire experiment. The table is sorted by mean jumps, highest to lowest. Immediately apparent is the surprising result of the ACM, an under-performing AVM in other respects, having the fewest average jumps per search, narrowly besting the combined searches. The NOM on the other hand, was the worst performing. Recalling that the ACM performs markedly worse in terms of overall actions and actions per second than the consecutive metrics, it stands to reason that its relative lack of jumping has to be countered by a propensity for faster, more frequent movement adjustments. None of the searching using consecutive action based AVMs recorded more than 2 jumps at any point in the experiment.

Completion Rate

Not every randomly selected start and end point is possible in our dataset. The original levels for the game assumed consistent left to right movement, and as such there are instances where the player is unable to return to the left, such as a large staircase the player can drop down but not jump back up to. Additionally, some levels feature ceilings, which can enclose spaces and make pathing strictly impossible. When such paths are selected, the algorithm reaches a predefined timeout point (5 seconds of calculation time in our experiment) and reports a failed search. Such paths are indistinguishable from timeouts caused by complex paths, or by our algorithm's complexity potentially being increased by our modifications. However in such cases, there will be a discrepancy between the searches in that some will complete a path while others will fail. Recall that in order to ensure fair comparisons, any path selected which was not completed by all searches was discarded, but we were interested in seeing how much of a discrepancy there was between default and modified behaviour.

In theory the default search should have the highest completion rate, because it is unconstrained relative to all of our AVMs and only selects the fastest path without regard for any human impositions. Additionally every AVM is a multiplier on top of the default behaviour, meaning for equal paths expanded would require more CPU cycles to complete. However as previously noted, we have observed greedy-like behaviour from some of our metrics in preliminary analysis, and by theoretically limiting the actions selected to those that are human achievable, and reducing the potential number of nodes as observed, we may in fact increase the completion rate of modified searches, as the levels themselves are obviously designed to be played by humans.

The completion rate of all searches is presented in Figure 5.14. The default search was able to complete approximately 45% of all searches. Recall that this accounts for both impossible paths as well as timeouts due to complexity. The ACM searches performed worse than default, with all other searches performing better. The NOM searches performed the strongest in this search.

We noted that the trend of the AVM performing poorly and the NOM performing well

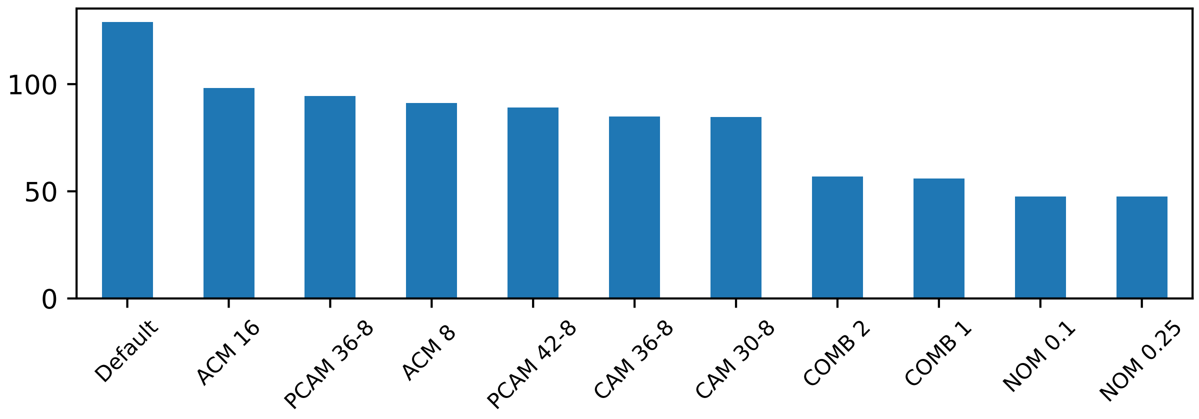
Table 5.2: Correlation between completion rate and efficiency metrics

	Completion Rate
Calculation Time	-0.81
Nodes per ms	0.78
Nodes Expanded	-0.65

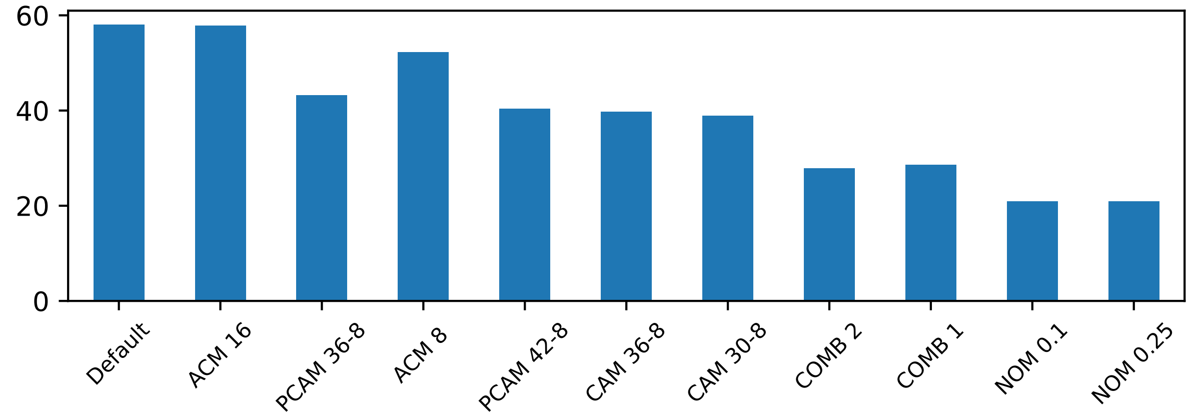
was very similar to the results of the calculation times. Examining further we computed the correlation and found a strong correlation between calculation time and completion rate, presented in table 5.2. It follows that any discrepancies in the completion rate are simply a result of the relative difference in performance between algorithms and likely not a result of our efforts to create human achievable paths specifically.

Figure 5.12: Efficiency Parameter Graphs

(a) Mean Nodes Expanded per Search



(b) Mean Calculation Time per Search



(c) Nodes per Milisecond per Search

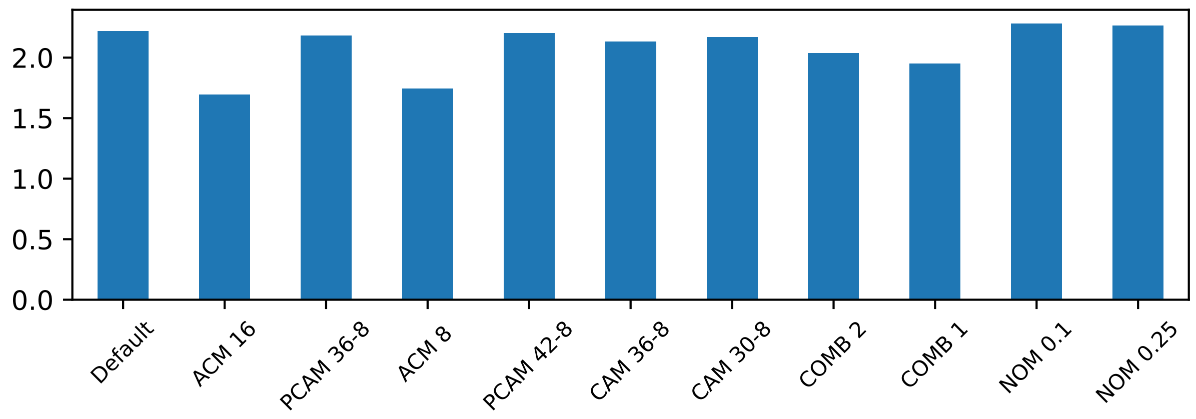
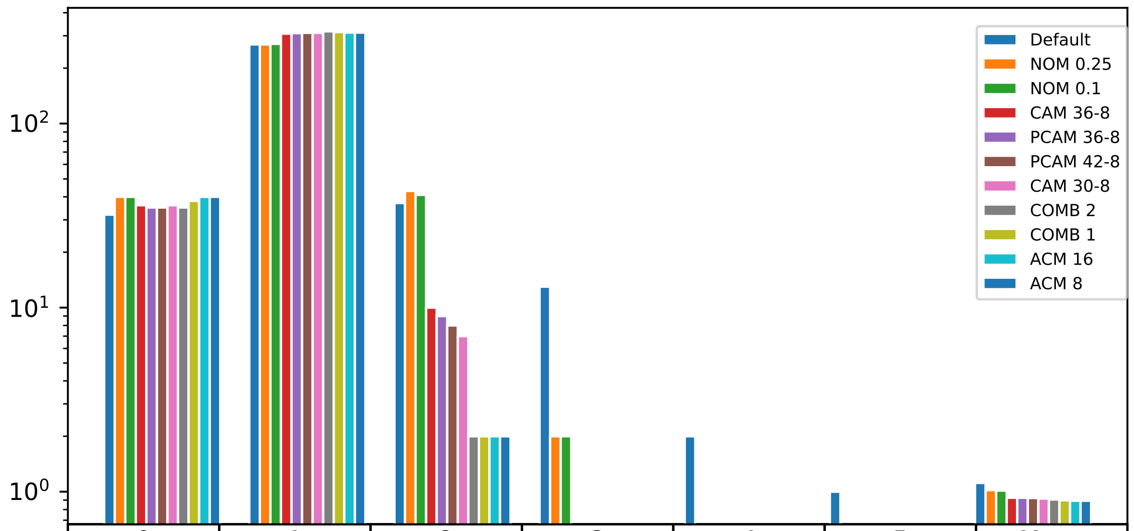
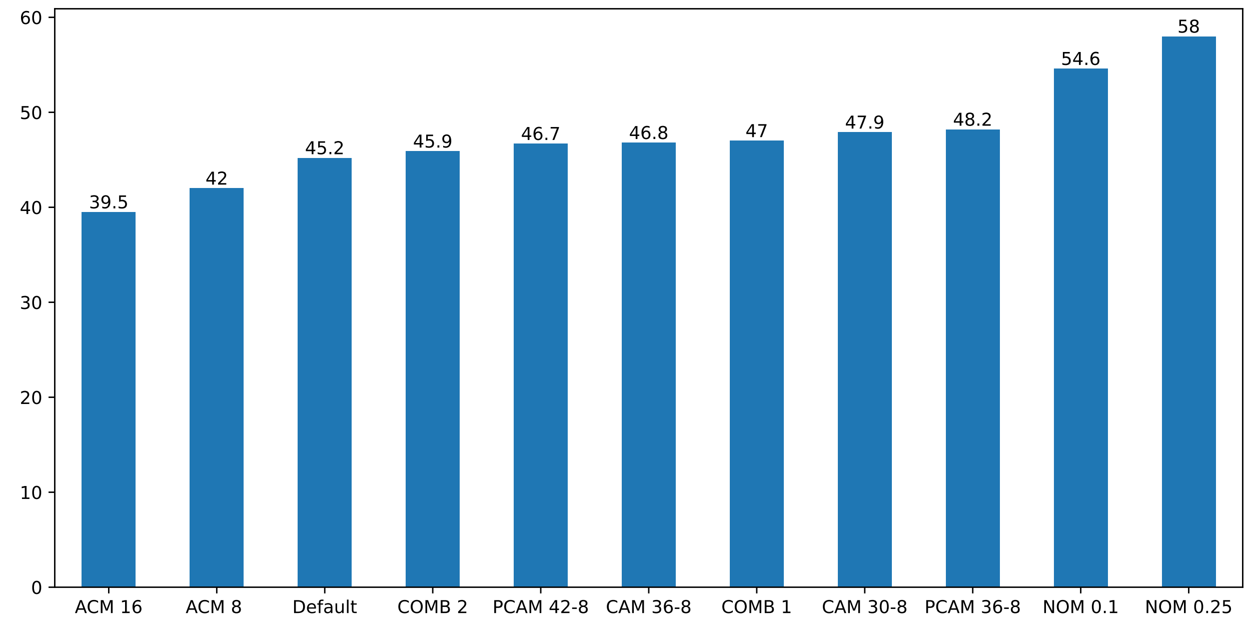


Figure 5.13: Number of jumps performed per-search across all searches



	0	1	2	3	4	5	Mean
Default	32.0	269.0	37.0	13.0	2.0	1.0	1.116
NOM 0.25	40.0	269.0	43.0	2.0	0.0	0.0	1.02
NOM 0.1	40.0	271.0	41.0	2.0	0.0	0.0	1.014
CAM 36-8	36.0	308.0	10.0	0.0	0.0	0.0	0.927
PCAM 36-8	35.0	310.0	9.0	0.0	0.0	0.0	0.927
PCAM 42-8	35.0	311.0	8.0	0.0	0.0	0.0	0.924
CAM 30-8	36.0	311.0	7.0	0.0	0.0	0.0	0.918
COMB 2	35.0	317.0	2.0	0.0	0.0	0.0	0.907
COMB 1	38.0	314.0	2.0	0.0	0.0	0.0	0.898
ACM 16	40.0	312.0	2.0	0.0	0.0	0.0	0.893
ACM 8	40.0	312.0	2.0	0.0	0.0	0.0	0.893

Figure 5.14: Completion rate of all searches



Chapter 6

Conclusion

Here we present an overview of the methodology and experiments, draw high level conclusions from our data, and propose further avenues for study.

6.1 Summary

From our analysis of previous pathfinding efforts in platform style games, we learned that the paths presented through AI pathfinding agents frequently play at a superhuman level. Attempting to use these paths as guidelines for players, especially ones of a lower skill level, would undoubtedly be difficult as they play at an unachievable level of skill.

Surveys on the subject as well as examination of the data from previous efforts identified three key areas in which using artificial pathfinding agent for guidance would generate impractical, unrealistic, and unplayable paths:

- Superhuman reaction times generate paths that are unplayable
- Jittery movement generates unrealistic looking play, and creates paths that are impractical to use as guides for players
- Lack of any danger avoidance creates paths that, were a human to attempt them, would frequently result in failure states

In order to create human achievable pathing through a videogame level, all three of these issues ought to be addressed.

To attempt to mitigate these issues, we have made several modifications to the A* algorithm. We elected to use a heuristic search based approach as one of our stated goals both for research and practical purposes was to be able to parameterize our results, such that various levels of skill and multiple versions of paths could be generated.

Our modifications to the A* algorithm are called *Action Value Modifiers* (AVMs) and we proposed four. These AVMs work by modifying the percieved path cost of the heuristic function through a series of real number valued multipliers. This approach allows the AVMs to be applied at various levels of intensity, as well as freely combined. The AVMs were designed with human skill in mind, taking approaches such as encouraging the algorithm to prioritize taking no actions, or increasing the perceived length of paths that make multiple actions in a short amount of time.

In order to evaluate the paths generated with regards to our goals, we selected 5 key metrics. Three of these metrics were designed to track the frequency of actions performed by the search, analogous the reaction time and physical speed that would be required for a human being to recreate the path. The other two of these metrics test what we call *Robustness*, in which attempts to follow the original path are reproduced but with the timing of actions shifted slightly, emulating noise introduced by a human player attempting to follow the path. We tracked the amount by which these perturbed paths deviate from the original, as well as the frequency at which they reach failure states. Producing a path that requires fewer actions, and is more robust to noise will address the issues we identified in analysis and will, we argue, generate paths more appropriate for use in guiding human players through a level.

Using our own videogame engine, we imported level data from the original Super Mario Brothers to act as a benchmark, and through visual analysis selected a range for each AVM that we believe best represents its ability to achieve our goals. With these parameter ranges selected we ran two sets of experiments. The first was run on a range of parameters for each individual AVM, comparing various intensities of our modifiers against default A* search behaviour. Selecting the two best performing values from each, we ran a second experiment, wherein thousands of randomly selected start and end points were chosen from within our level corpus. A search was performed on each path using all of our selected AVMs, including two searches using a combination of all the best performing parameters. These results were once again compared to the default A* search algorithm.

6.2 Conclusions

The best performing AVMs in our experiment performed well by every metric we had established for them. In particular, we found the AVMs in which consecutive actions performed within a short time frame were punished to far outperform the others, though the searches performed using a combination of all of the metrics had the strongest general showing.

Our best performing AVMs were able to decrease the total number of actions (analogous to button presses) required in total by 60%, decrease the actions per second by over 60%, and drastically reduce the amount of "superhuman" actions required by increasing the shortest amount of time between consecutive actions by over 350% on average.

In terms of robustness we were able to show a reduction in the variance of the perturbed paths final endpoint by 60%, and found that our best performing AVMs only reached failure states 30% of the time that the default search algorithm did when the paths were subjected to noise.

Looking at the absolute values of our timing improvements, the minimum amount of time between two button presses was increased from 10.5 frames to 37, or 167ms to 616ms. As the average human reaction time is considered to be 250ms, we believe this change clearly demonstrates the extent to which our AVMs have modified our paths to be more useful to guide human players.

None of our searches significantly increased the amount of time required to follow their paths, with the longest AVM only increasing time taken by less than 2%, which clearly demonstrates our new paths not to simply be longer and slower, but rather optimized to perform with human reproduction in mind, something that pathfinding algorithms do not generally take into account.

Additionally, the computational increase generated by our modifications was negligible, it seems that by incorporating human informed timing information into the A* node selection process, every one of our metrics was able to outperform the default algorithm in terms of runtime efficiency.

6.3 Future Work

The most promising area of follow-up research would be to apply the same AVM principles to pathfinding in other styles of videogame. Pathfinding is ubiquitous in many genres of videogames, and none of our proposed modifications to the algorithm have to do with platforming specifically. Indeed, this approach should work equally well in a 3 dimensional environment, though work would likely have to be done to incorporate the modern use of navigation meshes.

Another area that warrants investigation is in the selection of the parameterizations of the AVMs. Our selection process was performed initially through realtime human investigation of the parameterizations, and as such more optimal values may have eluded us. Because the AVMs are simply a series of real values numbers, they can be thought of as N-dimensional vector spaces. Optimization algorithms such as gradient descent could theoretically be used to find the highest performing values far better than our approach of grid searching through a stepped series of parameters.

Our tests were all performed within the level corpus of Super Mario Brothers, and it is entirely possible that while our parameter selection produces excellent results therein, it may be that different levels require differently valued AVMs to achieve similar performance, even if the physics and other mechanics of the game do not change.

Finally we are excited at the prospect that our parameterizable AVMs may be used to autonomously judge the difficulty of a video game level. By aggressively parameterizing AVMs and recording their performance or ability to complete a level, it may be possible to assign an objective numeric value to the difficulty required for a human being to complete the level. This would have obvious use in industry as automated QA testing becomes more commonplace.

Bibliography

[1] Alexander Braylan et al. “Frame Skip Is a Powerful Parameter for Learning to Play Atari”. In: *AAAI Workshop: Learning for General Competency in Video Games*. 2015.

[2] J. Bycer. *Game Design Deep Dive: Platformers*. CRC Press, 2019. ISBN: 9780429560576. URL: `https://books.google.ca/books?id=fdCwDwAAQBAJ`.

[3] Lin Chen and Damminda Alahakoon. “NeuroEvolution of augmenting topologies with learning for data classification”. In: *2006 International Conference on Information and Automation*. IEEE. 2006, pp. 367–371.

[4] T.H. Cormen et al. *Introduction to Algorithms, third edition*. The MIT Press. MIT Press, 2009. ISBN: 9780262258104. URL: `https://books.google.ca/books?id=F3anBQAAQBAJ`.

[5] Patrick Diskin. “Nintendo Entertainment System Documentation”. In: *Tokyo: Nintendo* (2004).

[6] Elijah Emerson. “Crowd pathfinding and steering using flow field tiles”. In: *Game AI Pro 360: Guide to Movement and Pathfinding*. CRC Press, 2019, pp. 67–76.

[7] Frederik Frydenberg et al. “Investigating MCTS modifications in general video game playing”. In: *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. 2015, pp. 107–113. DOI: `10.1109/CIG.2015.7317937`.

[8] Nobuto Fujii et al. “Autonomously Acquiring a Video Game Agent’s Behavior: Letting Players Feel Like Playing with a Human Player”. In: vol. 7624. Nov. 2012, pp. 490–493. ISBN: 978-3-642-34291-2. DOI: `10.1007/978-3-642-34292-9_42`.

[9] Nobuto Fujii et al. "Evaluating Human-like Behaviors of Video-Game Agents Autonomously Acquired with Biological Constraints". In: *Advances in Computer Entertainment - 10th International Conference, ACE 2013, Boekelo, The Netherlands, November 12-15, 2013. Proceedings*. Ed. by Dennis Reidsma, Haruhiro Katayose, and Anton Nijholt. Vol. 8253. Lecture Notes in Computer Science. Springer, 2013, pp. 61–76. DOI: 10.1007/978-3-319-03161-3_5. URL: https://doi.org/10.1007/978-3-319-03161-3%5C_5.

[10] Stuart Golodetz. "Automatic navigation mesh generation in configuration space". In: *Overload* 117 (2013), pp. 22–27.

[11] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[12] Juan Ortega et al. "Imitating human playing styles in Super Mario Bros". In: *Entertainment Computing* 4.2 (2013), pp. 93–104. ISSN: 1875-9521. DOI: https://doi.org/10.1016/j.entcom.2012.10.001. URL: https://www.sciencedirect.com/science/article/pii/S1875952112000183.

[13] Diego Perez-Liebana et al. "General video game ai: A multitrack framework for evaluating agents, games, and content generation algorithms". In: *IEEE Transactions on Games* 11.3 (2019), pp. 195–214.

[14] S. Pettus et al. *Service Games: The Rise and Fall of SEGA: Enhanced Edition*. Smashwords Edition, 2013. ISBN: 9781311080820. URL: https://books.google.ca/books?id=DbFxAgAAQBAJ.

[15] Luong Huu Phuc, Kanazawa Naoto, and Ikeda Kokolo. "Learning human-like behaviors using neuroevolution with statistical penalties". en. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. New York, NY, USA: IEEE, Aug. 2017, pp. 207–214. ISBN: 978-1-5386-3233-8. DOI: 10.1109/CIG.2017.8080437. URL: http://ieeexplore.ieee.org/document/8080437/ (visited on 04/05/2022).

[16] Luong Huu Phuc, Kanazawa Naoto, and Ikeda Kokolo. “Learning human-like behaviors using neuroevolution with statistical penalties”. en. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. New York, NY, USA: IEEE, Aug. 2017, pp. 207–214. ISBN: 978-1-5386-3233-8. DOI: `10.1109/CIG.2017.8080437`. URL: `http://ieeexplore.ieee.org/document/8080437/` (visited on 04/05/2022).

[17] Steven Rabin. *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. CRC Press, 2015.

[18] Craig W Reynolds. “Flocks, herds and schools: A distributed behavioral model”. In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, pp. 25–34.

[19] Craig W Reynolds et al. “Steering behaviors for autonomous characters”. In: *Game developers conference*. Vol. 1999. Citeseer. 1999, pp. 763–782.

[20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. 2010.

[21] Noor Shaker et al. “The turing test track of the 2012 Mario AI Championship: Entries and evaluation”. In: *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. 2013, pp. 1–8. DOI: `10.1109/CIG.2013.6633634`.

[22] Adam James Summerville et al. “The VGLC: The Video Game Level Corpus”. In: *CoRR* abs/1606.07487 (2016). arXiv: `1606.07487`. URL: `http://arxiv.org/abs/1606.07487`.

[23] Jonathan Sykes, J Rutter, and J Bryce. “A player-centred approach to digital game design”. In: *Understanding digital games* (2006), pp. 75–92.

[24] Sila Temsiririrkkul et al. “Survey of How Human Players Divert In-game Actions for Other Purposes: Towards Human-Like Computer Players”. en. In: *Entertainment Computing – ICEC 2017*. Ed. by Nagisa Munekata, Itsuki Kunita, and Junichi Hoshino. Vol. 10507. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 243–256. ISBN: 978-3-319-66714-0 978-3-319-66715-7. DOI:

10.1007/978-3-319-66715-7_27. URL: https://link.springer.com/10.1007/978-3-319-66715-7_27 (visited on 04/05/2022).

[25] Sila Temsiririrkkul et al. "Survey of How Human Players Divert In-game Actions for Other Purposes: Towards Human-Like Computer Players". en. In: *Entertainment Computing – ICEC 2017*. Ed. by Nagisa Munekata, Itsuki Kunita, and Junichi Hoshino. Vol. 10507. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 243–256. ISBN: 978-3-319-66714-0 978-3-319-66715-7. DOI: 10.1007/978-3-319-66715-7_27. URL: https://link.springer.com/10.1007/978-3-319-66715-7_27 (visited on 04/05/2022).

[26] Julian Togelius, S. Karakovskiy, and Robin Baumgarten. "The 2009 Mario AI Competition". In: Aug. 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586133.

[27] C.W. Totten. *Architectural Approach to Level Design: Second edition*. CRC Press, 2019, pp. 49–53. ISBN: 9781351116282. URL: https://books.google.ca/books?id=PQqWDwAAQBAJ.

[28] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (Oct. 1950), pp. 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433.

[29] Wouter Van Toll et al. "A comparative study of navigation meshes". In: *Proceedings of the 9th International Conference on Motion in Games*. 2016, pp. 91–100.