The Effects of Human-like Modifications to Heuristic Action Evaluation in Video Game Pathfinding

Robert Bishop
Memorial University of Newfoundland
St. John's, Canada
r.bishop@mun.ca

David Churchill
Memorial University of Newfoundland
St. John's, Canada
dave.churchill@gmail.com

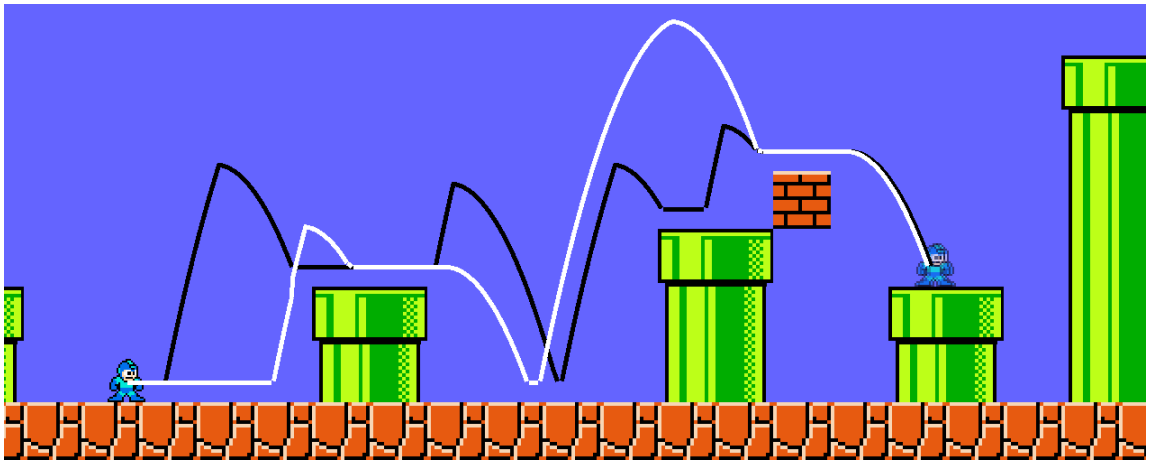


Figure 1: Two different paths to the same goal produced by our system. Click the image for a video demo.

ABSTRACT

We present a series of parameterizable modifications to heuristic evaluation of actions in the A* algorithm, designed to create more human-like and dexterity-robust paths through games in the 2 dimensional platformer style. We attempt to create paths at various levels of player skill by imposing constraints onto the timing and duration of actions designed to mimic human reaction times and ability. We show that these action value 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.

CCS CONCEPTS

• **Computing methodologies** → **Heuristic function construction**.

KEYWORDS

Video Games, Pathfinding, A*

ACM Reference Format:
Robert Bishop and David Churchill. 2022. The Effects of Human-like Modifications to Heuristic Action Evaluation in Video Game Pathfinding. In *FDG '22: Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22), September 5–8, 2022, Athens, Greece.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555858.3555888

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[1] and MarIO neural network[2], which have garnered millions of views online. These videos are popular not only for the impressiveness of the technical challenge, but 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. A surge in interest in AI-aided play lead to the creation of the Mario AI championship, a competition which ran from 2009 to 2013 [Togelius et al. 2013] in which participants created their own automated agents and competed against one another. Impressive as these agents are, their performances are only made possible by

FDG '22, September 05-08, 2018, Athens, Greece
2022. ACM ISBN 978-1-4503-9795-7/22/09…$15.00
https://doi.org/10.1145/3555858.3555888

[1] https://www.youtube.com/watch?v=DlkMs4ZHHr8
[2] https://www.youtube.com/watch?v=qv6UVOQ0F44

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[3], and paths that are only possible via superhuman input precision are called "TAS-only" strategies - while technically possible, they are infeasible 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.

Existing AI pathfinding systems however, as previously mentioned, typically produce super-human paths that would be impossible to implement by humans, especially those just learn to play the game. The reasoning for this is due to the fact that these pathfinding systems use heuristic action evaluations that optimize paths for either distance or speed, with no care for the relative *difficulty* of its implementation, due to the fact that computer players have perfect input precision. In this paper, 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, in hopes 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 limiting the frequency and number of actions a player would be expected to perform. In 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 recreation, and even attempt to quantify a level of safety based on how likely these perturbed paths are to reach failure states.

2 BACKGROUND AND RELATED WORK

2.1 A* for Real-time Games

The A* algorithm is a popular and industry-wide algorithm for pathfinding in general cases [Rabin 2015]. The core of the algorithm is the combination of a priority queue and a heuristic function to allow the algorithm to operate as a best-first search. In most graph-based environment path-finding problems, actions are limited to movement in the environment, and as such, action cost is measured as the distance travelled by the action. Most modern video game path-finding systems are split into two stages: the path-finding stage, and the path-following stage (locomotion). Modern games typically use a navigation mesh abstract representation of the environment, perform A* path-finding on that abstracted navmesh graph, and then once a path is found they attempt to follow that abstract path in the real game space.

A very important difference in this paper from traditional path-finding systems is that our search space is not abstracted to a navigation mesh structure, and is instead performed in the actual action space of the user input of the player. Actions such as moving, jumping, and shooting are all valid within our path-finding search. For example, the shortest path in a level may involve running, jumping off a ledge, and then shooting a hole in a wall that the player can then traverse through, all while dodging enemies that may be blocking our path. As such, we can no longer use a simple distance function as a heuristic evaluation for the A* algorithm, and instead we must use an estimate of the *game time* (frames) it will take the user to get to a location. The simplest of these functions could just take the distance to a goal location and divide it by the player's maximum speed in order to get an estimate of the time it takes to get to goal. It should be noted that due to the complex game rules of real-time platformers such as jumping, gravity effects, and move speed altering power-ups, it can be very difficult to guarantee that our heuristic functions are *admissible*, a quality required for A* to produce optimal paths. We are therefore operating under the assumption for the rest of this paper that strict optimality (in terms of game time to each a goal) is not guaranteed. Time-optimal paths can be achieved if a heuristic evaluation of 0 is used, however this leads to a trade-off in performance as with any heuristic search system - with the 0 heuristic providing no guidance for the search while a very poor heuristic provides poor overall paths.

This type of real-time search in video games presents additional challenges over traditional graph-based environments as well, since the only viable analog for movement occurs when the game updates its state. For a typical video game, this occurs 30-60 times a second. The search tree therefore grows extremely quickly even when attempting to calculate just a few seconds of movement. In order to combat this increase in search space most real-time video game search functions utilize what is known as *skip frames*, which represent a batching of time in the search space. Once an action has been issued, the resultant child node is generated not by advancing the game forward one frame, but by a given of skip frames. This leads to an additional parameterized trade-off in our search system, with lower skip frame counts yielding longer but more fine-grained paths, and higher skip frame counts yielding faster search times but very coarse grained solutions with long delays between actions. If however these skip frames are tuned to mimic human-like reaction times, they can greatly reduce the complexity of the search space while still retaining adequate performance [Braylan et al. 2015].

2.2 Human-like Pathing

While optimal paths can indeed be achieved, their super-human requirements for perfect dexterity in their implementation can be easily discerned by most human players [Fujii et al. 2012], and as such are not viable for tutorial or guidance systems in games. Attempts to create more realistic human-like agents have focused

[3] https://tasvideos.org/

on areas outside of heuristic search, be it through neuroevolution [Ortega et al. 2013] or deep learning [Phuc et al. 2017]. Additionally, most attempts to replicate human behavior have focused on replicating human imperfections, such as allowing actions unrelated to the goal state of the agent, or waiting for arbitrary lengths of time [Temsiririrkkul et al. 2017].

Past experiments on the subject have used subjective analysis and surveying to judge the perceived quality of the agents. In 2012 a Mario "Turing Test" was performed [Shaker et al. 2013] in which participants were tasked with creating, and judging, the quality of AI agent's ability to mimic a human playing style. This test introduced a formal series of metrics, however they were specific to the Super Mario Bros. game, and several of the metrics relate to specific game elements such as enemies and coins, not all of which are present in all games. Of the top 3 entries in the competition, two used influence maps and the third used a neural network.

3 METHODOLOGY

Most of the novelty of our system involves modifications to the standard time-optimal heuristic function previously discussed. These modifications disincentivize the production of paths that would exceed the limits of human ability. We achieve this by focusing on modifiers that limit the frequency of actions required to reach the goal. These **Action Value Modifiers** (AVMs) are described in section 3.3, whose parameters are stored in external configuration files, which allow us to easily run experiments of many different types.

Searches are performed in our custom C++ game engine in real time, with all player inputs representing possible actions. Having complete control of this custom game engine allows us to copying, rewind, and fast-forward any given game state, which facilitates the use of heuristic search algorithms such as A*. The result of these searches generate path data as a replay, described in section 3.2, which can be saved as files for analysis or review. With a search completed, our system runs analysis on the replays, tracking a number of metrics we have selected which we believe best indicate that the modified path has become more human-like. These metrics are mostly related to the concept of action input timings, and are discussed in depth in section 4.1

Using this system allows us to test parameters quickly to find ranges that look visually appropriate, before defining a larger range of parameters we want to test experimentally. Data is tracked as the searches are performed and both output visually and saved to disk for further analysis.

3.1 Custom 2-D Game Engine

Our 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 Multumedia Library (SFML) [4] 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:

- Completely deterministic
- Capable of generating and reading replays to recreate play
- 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.

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 [Summerville et al. 2016].

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 new player position 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. 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 that 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 resultant position, it would retain all functionality in a new game environment with no changes required to the search code whatsoever.

3.2 Searches and Replays

Central to our methodology are the concepts of *Searches* and *Replays*. The former being a class of functions dedicated to generating the latter, which itself is simply a data class.

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 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 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

[4] https://www.sfml-dev.org

replay.txt

```
0    START-RIGHT
18   START-JUMP
30   END-JUMP
48   END-RIGHT
```

Figure 2: A sample replay file of a search conducted with a frame skip value of 6. In this replay, the player would start by immediately running to the right, pressing the jump button at frame 18 for 12 frames, and then stopping 12 frames later.

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 over to both minimize the number of actions required in the replay file, as well as 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. While many aspects of our search algorithm are paramaterizable, we have elected to keep a constant value on frame skipping. Frame skipping is one of the standard techniques for game state exploration and pathfinding and has a documented history. 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 is "locked in" for the next n frames. While this metric has been used in some other simulations of human-like behaviour it has some notable problems, especially when expanded to large values to simulate human reaction time. While Deepmind's Atari DQN network used a frame skip value of 4 [Mnih et al. 2013], after experimental testing we have opted to use a more lenient 6 frames on all subsequent tests. Given our game engine runs at a consistent 60fps this also gives us a round value of 10 actions per second, which is comparable to the dexterity of experienced players. Keep in mind that 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 this paper is that of action value modifiers, which act as modifiers on the heuristic cost function of actions in our search algorithm. 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 an *action change* by our definitions. 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 actual actions. It is a fairly naive way to incentivize reducing the number of actions, and at large values can result in paths that perform poorly with our chosen metrics. Because this action is by far the most common in normal human play of most video games, setting the NOM to lower values results in greedy search-like behaviour, and usually greatly reduces the number of nodes expanded as well as calculation time for a given search compared to default search behaviour, however this is not the primary concern of our efforts.

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 action changes. 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 more closely emulates 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 of $f(n) = g(n) + h(n)$, 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, or 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 just the

current frame count of the game state, with individual action costs not really having any relevance.

Therefore, each of our AVMs act as multipliers on the value of $h(n)$, 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 NOM=0.7 and CAM=1.3 then our A* search instance would use $f(n) = g(n) + h(n) * 0.7 * 1.3$.

4 EXPERIMENTS AND RESULTS

We conducted a number of experiments comparing searches using variations of our modifiers against a default (unmodified) A* implementation. Specifically the tests work as follows:

- Define a number of custom search instances through various parameters of our modifiers.
- Randomly select a level from our level collection
- Randomly select two walkable tiles a sufficient distance from each other, designating one as the start point and one as the goal point
- Run each instance of the modified A* algorithms, including the default, tracking metrics for all individually
- Repeat for a set number of iterations

4.1 Evaluation Metrics

Our evaluation metrics center around the concept of actions, or more specifically changes in action state, which attempt 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 is not considered to change the action state, as a human being holding the button to move right requires no 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.

During the experiment, several metrics are tracked per each instance of randomized start/goal search, and are then aggregated to produce a final value representing the model's general performance. Of particular interest we present the following metrics and their rationale:

- Total number of action changes performed: "How many inputs are required to be pressed to complete this path?"
- Mean time between all action changes: "What is the average time between pressing inputs required to complete this path?"
- The mean of minimum time between action changes per path: "What is the shortest time between two consecutive input presses required to complete this path?"

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 can be considered analogous to creating more human-like pathing.

4.1.1 *Robustness and Sensitivity Analysis.*

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 track robustness using a form of sensitivity analysis. Once a path 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.

Algorithm 1: Perturbing Replays

```
Data: R
Result: RP = [R_1, R_2, ...R_n]
begin
    RP ← []
    for i ∈ [−20, 20]; i ∈ ℤ do
        foreach A in R do
            R_i ← R ;                          /* copy replay */
            foreach AP in R_i do
                if AP_t >= A_t then
                    AP_t += i
            RP.insert(R_i)
```

A brief overview of our perturbation algorithm can be seen in Algorithm 1. Each search being considered by our experiment generates a replay R which we will use to create a number of perturbed replays 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 End Distance sums up the final position of all instances of the perturbed paths, averaging the distance by which all differ from the original end position.
- Mean Failure Rate tracks the percentage of instances which reach a premature end state, falling into a pit in our case. If no failure states are reached by any search, this value is ignored for that instance.

In general these metrics attempt to track how "safe" a path is (how robust it is to small deviations in input) . If there is a particularly difficult jump that must be performed with precise timing to progress, then there will be a large deviation in the end positions of perturbed paths, and the Mean End Distance will be high. Similarly if a path skirts dangerously close to a game-over state (pitfalls or enemies), then the failure rate of the perturbed paths will be quite high.

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.

Table 1: Best Performing Parameters

AVM	Weight	Total Actions	Mean End Distance
NOM	0.75	91.9%	95.0%
ACM	8	82.2%	96.6%
CAM	36, 8	75.9%	72.0%
PCAM	36, 8	72.5%	71.2%

4.2 Selecting Best Performing Parameters for Each Modifier Individually

Before testing the various modifiers against each other, we had to select specific values for the parameters. The NOM and ACM modifiers both define themselves through a single floating point multiplier to path cost, while CAM and PCAM consist of the same path multiplier in addition to an integer frame window. Performing an exhaustive search of the potential space would be intractable for this paper, and as such a range of potential values was determined subjectively using real-time searches and visual analysis. With these ranges determined, a series of tests were run, comparing all parameters against each other as well as the default search. While we tracked many data points, for brevity and clarity we will present the total action changes as well as the perturbed end distance for each section. The former representing, broadly, the complexity of the path and the latter representing its safety. The best performing searches are presented in Table 1 with the values given as their percentage of default behavior. For both metrics a lower number indicates better performance, as we want paths that take fewer actions, and deviate less when noise is introduced.

4.2.1 NOM.

For the NOM we ran tests comparing values of 0.75, 0.5, 0.25 and 0.1. As this number represents a value multiplier on heuristic length, lower values make not taking an action more favorable, and be considered to be increasing the "strength" of the parameter. While end distance increased fairly linearly, the total number of actions for all values decreased significantly compared to default, but barely increased with lower values. A value of 0.75 was selected.

4.2.2 ACM.

For ACM we tested values of 2, 4, 8 and 16. As with the NOM, all tests performed fewer actions than default, however the advantage of higher values decreased significantly, with the change from 8 to 16 not affecting the number of actions at all, and actually increasing the mean end distance of perturbed paths by 0.1%. As such a value of 8 was selected.

4.2.3 CAM.

For the CAM we had to select a range for the frame window as well as the multiplier. Given our searches are using a skip frame value of 6, we elected to test frame windows of 6, 12, 18, 24, 30, and 36. For multiplier values we tested 2, 4 and 8. Every combination was tested. As an overall trend the higher values for the frame window parameter performed better, and higher multipliers performed better as well. The search with a 30 frame window and a multiplier of 8, performed identically to the search with a 36 frame window with a multiplier of 8 in terms of actions and survival rate, and as such the latter was selected.

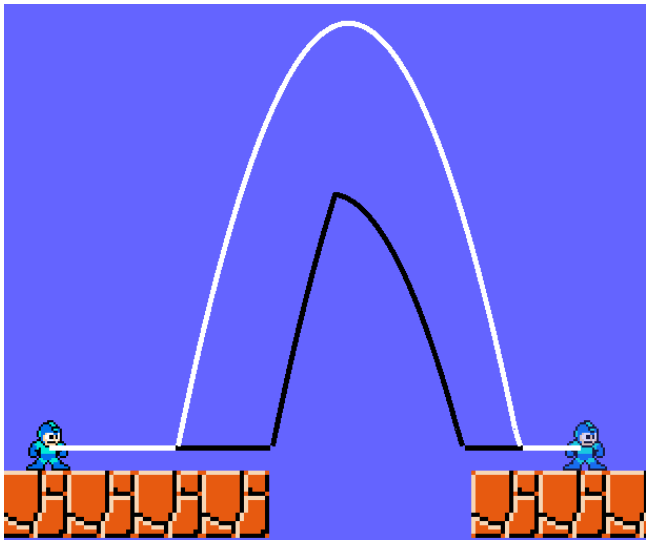


Figure 3: Example of a "Robust" jump arc. Default A* search results in the black path, while AVM search results in the white path, avoiding dangerous edges.

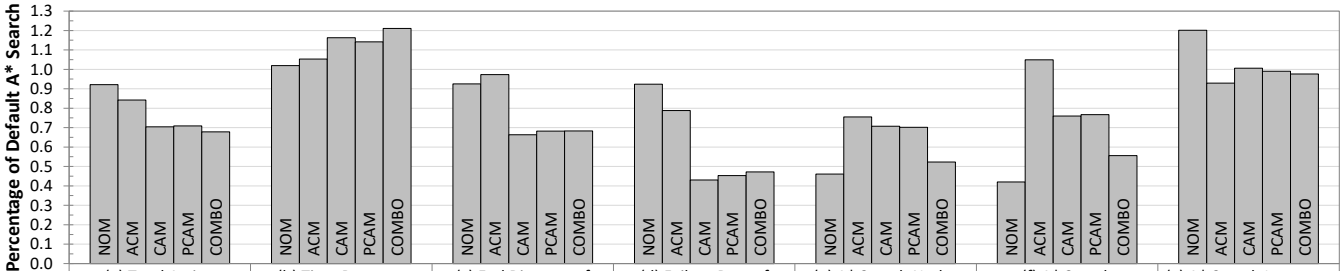
4.2.4 PCAM.

For the PCAM we found better results with larger value for the frame window parameter compared to the CAM, likely due to the nature of the linear falloff of the multiplier. As such our frame windows were selected from the range of 12 to 42, increasing by 6 as before. For multipliers we again used values of 2, 4 and 8. Every combination was tested and curiously as before the two most extreme values: frame window 36 with multiplier 8, and frame window 42 with multiplier 8, performed near identically. The larger 42 frame window gave a slight increase in mean end distance for its perturbed paths, and as such a frame window of 36 with a multiplier of 8 was selected.

4.3 Comparing Best Performing Searches

With the best individual values selected, we ran an experiment where all searches would be compared using the same paths. While the individual tests gave us a sense of each modifiers performance relative to the default, each experiment was run on its own set of randomized paths, and as such some of the data might not be comparable. The methodology for this second experiment was the same as the previous, but instead of testing a single modifier at multiple values, we selected the previous best performing candidates to compare them against each other. We also created a "combination" value which used all the modifiers of the best performing candidates in tandem. 1000 iterations were performed, testing each algorithm using the same randomized start and end points on randomly selected levels from Super Mario Bros.

- NOM: Set to 0.75 to undervalue the heuristic of taking no action by 25%.



	(a) Total Action Changes	(b) Time Between Action Changes	(c) End Distance of Perturbed Paths	(d) Failure Rate of Perturbed Paths	(e) A* Search Nodes Expanded	(f) A* Search Completion Time	(g) A* Search Instances Completed
NOM	0.921	1.019	0.925	0.924	0.461	0.420	1.201
ACM	0.842	1.053	0.973	0.788	0.755	1.049	0.929
CAM	0.704	1.163	0.663	0.430	0.707	0.760	1.006
PCAM	0.709	1.142	0.682	0.453	0.702	0.767	0.990
COMBO	0.678	1.211	0.683	0.472	0.523	0.556	0.976

Figure 4: Experimental results for Action Value Modifier A* Search vs. Default A* Search. All values are shown as a percentage (0.9 = 90%) of the baseline Default A* Search result, averaged over 1000 randomized search instances for each experiment.

- ACM: Set to multiply the heuristic value of any action that changes the player's action state by 8.
- CAM: Set to multiply the heuristic value of any action occurring 36 frames within a previous action by 8.
- PCAM: Uses the same values as the CAM but linearly decreases the penalty as described previously.
- COMBO: Applies all the above modifiers together, using the progressive version of the consecutive action penalty.

4.4 Results

The results for our AVM experiments can be seen in Figure 4, please refer to this Figure for all future results discussion.

4.4.1 Actions.
Action changes in our replays are analogous to individual human inputs, and so ideally we would like to see fewer total action changes performed. In Figure 4a we can see that each of the AVMs produced fewer total actions than the Default A* search (values < 1), with CAM being the best performing of the individual AVMs, with the COMBO search being the best overall performer. A general trend can be observed of the ACM outperforming the NOM, and the two Consecutive Action variants and combination all performing near equally well when compared the the ACM. This shows us rather conclusively that use of the frame window penalty technique seems to be the best method for generating paths with fewer required actions.

4.4.2 Time Between Actions.
To make paths more human-feasible, we would also prefer to see a longer average time duration between action inputs. In Figure Figure 4b we again see that the AVM searches outperformed the Default A* search in this metric, with values all > 1. Once again CAM outperformed PCAM with the combination search being the best overall. This shows that not only do the frame windowing techniques create paths with fewer actions, but that the resulting actions are more evenly spaced. It is worth noting that in earlier tests on the individual parameters, high values of ACM in particular did not have a strong correlation between total actions and shortest duration. Meaning that while the paths had fewer actions in total, they would often be found in relatively short time frames.

4.4.3 Robustness Metrics.
For our robustness metrics we observed an even more dramatic trend of the windowed versions of the searches to be the best performers. For the mean end distance of the perturbed paths shown in Figure 4c, each AVM outperformed than the Default A* search with values < 1, with a dramatic decrease in end distance for our final 3 AVMs. This means that our AVMs produce paths which are more robust to action perturbations in terms of still arriving close to the original destination. As before, the CAM was the strongest performing of the individual parameters, with COMBO being the best performing overall.

For the failure rate of perturbed paths shown in 4d, we can see that our AVMs again performed better than the Default A* search, yielding paths that were more robust to action perturbations that would result in the player falling into a pit and dying. This result is the one metric that breaks the general trend, in that COMBO did not offer the best performance. Here the CAM offered the single best performance overall, with less than than half the failure rate of the default search. An illustration of the robustness of our AVMs for this metric can be seen in Figure 3. With properly tuned parameters this jump not only increases the average time between actions (Default: 34, Modified: 51), but as a result the path avoids the dangerous edges of the pit. Because the default black path waits until the last possible frame to jump, and lands as soon as possible, many of the perturbed paths that shift the jump action forward or backwards would reach a fail state. The modified jump has much more room for input error, and would be much safer for a human being to attempt to replicate.

4.4.4 Path Execution Time.
Also of importance to our search is the actual game time spent in traversing the paths produced by the various AVM search instances. Previously we estimated that we would see an increase in total travel time as we were expecting to trade path speed for other human-like qualities, but our experiments showed that the path

completion times were nearly identical to the Default A* search for all AVMs tested. The worst performing search on average of all the AVMs produced paths which took less than 1% more time to complete than the Default A* search, and due to this small difference the exact values were omitted from the results table.

4.4.5 *A* Search Metrics.*

When designing our AVM system we did not have a goal of improving overall search performance in mind, but we pleasantly found out later that they can in fact improve search times. In Figure 4e we can see that on average, each of our AVM searches resulted in far fewer node expansions than the Default A* search. This means that in general, fewer search iterations were performed, and (ideally) searches take less time to complete. While we cannot prove the reasoning behind this performance increase, our intuition is that by guiding the search toward a more constrained human-like action space likely ends up in searching fewer nodes, due to the implicit pruning of more dense sequences of action changes. We can see the results of actual wall-clock search times in Figure 4f, which unlike nodes expanded also factors in the calculation time of each AVM. We can see that each AVM search took significantly less time than the Default A* search with the exception of ACM, which on average took 5% longer. The average instance search time for the Default A* algorithm was 23.3 milliseconds. Each search instance was given a very lenient 5 seconds to complete before timing out, however our experiment searches took on average between 9-24ms per instance.

The final metric we measured was search instance completion rate. When performing our search instances, we randomly selected two points as the start and end points for the player. While this demonstrates the ability to find paths in various points in the environment, it is not guaranteed that a path actually exists between the start and end points. If the search failed to complete in time, it would simply be marked as incomplete. For the sake of fairness of comparison, if any of the search instances failed to complete a path, then that particular search instance was not used in calculating the previous metrics. We can see from the results in Figure 4g that there was not a significant difference between search completion rates for the Default A* search and the AVM searches, and we could not gain any insight from this metric.

5 CONCLUSION AND FUTURE WORK

In this paper we have introduced the novel idea of Action Value Modifiers, a set of heuristic action evaluations that when paired with the A* search algorithm produce paths that yield qualities that are more human-like than a baseline default A* search that only minimizes path completion time. Our experiments show the paths produced by our AVMs yield fewer total action inputs, space out those actions by longer periods of time, and result in paths that are more robust to action input noise. By perturbing the timing performed actions in the simulation of a path, we show that our techniques result in paths that reach failure states less often, and generally come closer to the end goal when compared to the baseline. We have also shown that not only do these paths have more desirable qualities, but they take less time to calculate on average than a default A* search.

There are many avenues we wish to pursue in the future for this work. Of particular interest would be the actual testing of these generated paths with human participants. While measuring various metrics does show that we have achieved our initial goals for this paper, the true test would be to have human players attempt to follow both sets of paths in a tutorial setting similar to Nintendo's Super Guide feature, and then comment on which they found to be more useful in enhancing their gameplay experience.

Additionally, we would like to extend our initial ideas of AVMs to create a system which can infer the human perceived difficulty of a path or even an entire level layout. Conceptually, by creating searches with more punishing modifiers to the timing of actions, we can better approximate the ability of less skilled players. Dynamically tuning the values of the search parameters on a path until it reaches a failure state could be a prospective source of automatically determining the skill level required to reach a goal. A longer term goal would then be to use this difficulty inference to creation of content of varying difficulties by combining it with techniques in the field of procedural content generation.

REFERENCES

Alexander Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. 2015. Frame Skip Is a Powerful Parameter for Learning to Play Atari. In *AAAI Workshop: Learning for General Competency in Video Games*.

Nobuto Fujii, Yuichi Sato, Hironori Wakama, and Haruhiro Katayose. 2012. Autonomously Acquiring a Video Game Agent's Behavior: Letting Players Feel Like Playing with a Human Player, Vol. 7624. 490–493. https://doi.org/10.1007/978-3-642-34292-9_42

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

Juan Ortega, Noor Shaker, Julian Togelius, and Georgios N Yannakakis. 2013. Imitating human playing styles in super mario bros. *Entertainment Computing* 4, 2 (2013), 93–104.

Luong Huu Phuc, Kanazawa Naoto, and Ikeda Kokolo. 2017. Learning human-like behaviors using neuroevolution with statistical penalties. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, New York, NY, USA, 207–214. https://doi.org/10.1109/CIG.2017.8080437

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

Noor Shaker, Julian Togelius, Georgios N. Yannakakis, Likith Poovanna, Vinay S. Ethiraj, Stefan J. Johansson, Robert G. Reynolds, Leonard K. Heether, Tom Schumann, and Marcus Gallagher. 2013. The turing test track of the 2012 Mario AI Championship: Entries and evaluation. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. 1–8. https://doi.org/10.1109/CIG.2013.6633634

Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santi Ontañón Villar. 2016. The VGLC: The Video Game Level Corpus. *CoRR* abs/1606.07487 (2016). arXiv:1606.07487 http://arxiv.org/abs/1606.07487

Sila Temsiririrkkul, Naoyuki Sato, Kenta Nakagawa, and Kokolo Ikeda. 2017. Survey of How Human Players Divert In-game Actions for Other Purposes: Towards Human-Like Computer Players. In *Entertainment Computing – ICEC 2017*, Nagisa Munekata, Itsuki Kunita, and Junichi Hoshino (Eds.). Vol. 10507. Springer International Publishing, Cham, 243–256. https://doi.org/10.1007/978-3-319-66715-7_27 Series Title: Lecture Notes in Computer Science.

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