# Automated Testing in Super Metroid with Abstraction-Guided Exploration

Ross Mawhorter
rmawhort@ucsc.edu
University of California Santa Cruz
Santa Cruz, CA, USA

Adam Smith
amsmith@ucsc.edu
University of California Santa Cruz
Santa Cruz, CA, USA

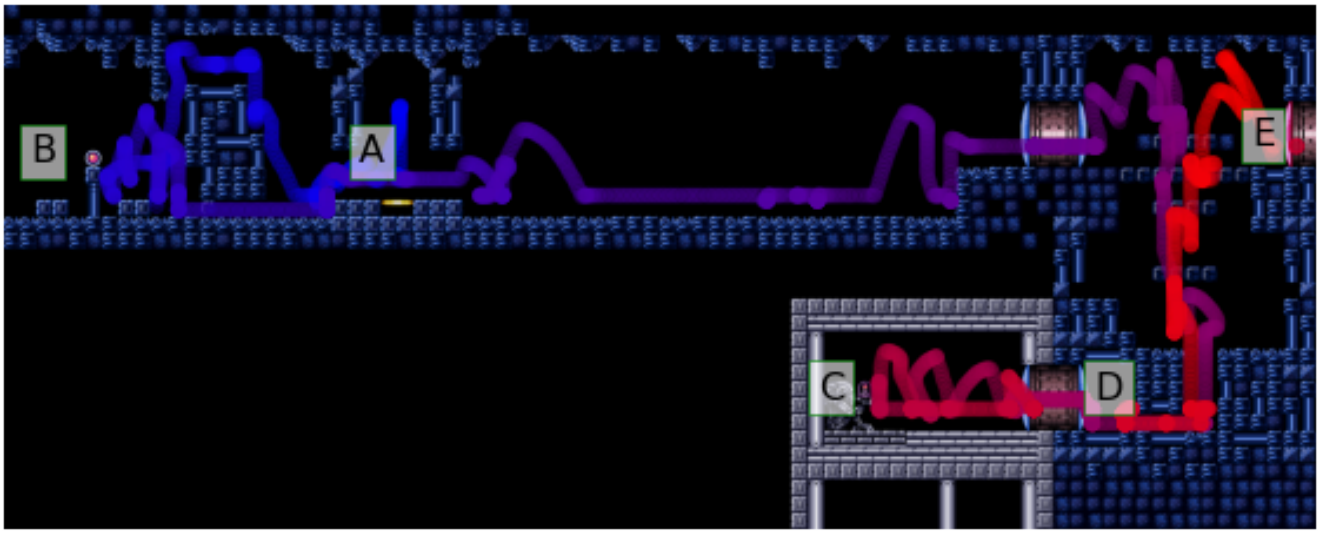


**Figure 1: Exploration waypoints and solution paths discovered using abstraction-guided exploration in Super Metroid. Color ranges from blue to red along the path to illustrate how the player backtracks. The total non-cutscene gameplay length is 3250 frames, compared with an estimated human expert gameplay of 2420 frames.**

## ABSTRACT

Machine playtesting systems often aim to demonstrate how to reach certain moments of play. To provide design feedback in a timely manner, they often internally rely on heuristic-guided search. However, there are many types of videogames for which sufficiently accurate heuristics are not available. We use an imperfect abstraction of an underlying game to define progress scores, and show that combining these scores yields a highly effective cell selection heuristic for use in the Go-Explore algorithm. We demonstrate the impact of this approach in automated gameplay for Super Metroid (involving mandatory item collection, destructible blocks, and backtracking) using a tile-based abstraction of the game that only models a small subset of the game's mechanics. Surprisingly, our abstraction guidance mechanism is able to explore this complex game several orders of magnitude more efficiently than past work with similar exploration methods in Montezuma's Revenge.


*FDG '23, April 12 2023, Lisbon, Portugal*

ACM ISBN 978-1-4503-XXXX-X/18/06…$0.00
https://doi.org/10.1145/3582437.3582447

## CCS CONCEPTS

• **Applied computing** → **Computer games**; • **Computing methodologies** → **Heuristic function construction**; *Abstraction and micro-operators.*

## KEYWORDS

machine playtesting, search heuristic, go-explore, state abstraction

**ACM Reference Format:**
Ross Mawhorter and Adam Smith. 2023. Automated Testing in Super Metroid with Abstraction-Guided Exploration. In *Foundations of Digital Games 2023 (FDG 2023), April 12–14, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3582437.3582447

## 1 INTRODUCTION

Testing is an important part of the game design process [11]. While some aspects of human playtesting focus on aesthetic concerns and players' emotional responses, another large focus of testing is ensuring that the game is free of embarrassing bugs [6]. Coverage of all of the interesting things a player can do in a game is often

achieved through the high-stress, low-pay labor of quality assurance (QA) workers. Automated testing systems (that allow human operators to set high-level coverage goals for algorithmic gameplay agents) offer to increase coverage and reduce time-to-feedback in the larger game design process [4, 27]. Often, this feedback details what states the player can reach after making the change. By looking at these outputs, a designer can be certain that the changes that they make do not break any internal invariants (for example, "the player picks up item X before they reach location Y"). Fig. 1 offers an example reachability report resulting from our work that shows *how* the player can reach a distant goal by collecting item required along the way.

In order to produce this type of feedback, automated testing systems often rely internally on some kind of search to explore the game's state graph [16, 31]. However, when the game's state graph is very complex, these methods may not be able to provide feedback quickly enough to be useful, as reaching the states that the developer cares about becomes less and less likely given a fixed compute budget. To solve this problem, search algorithms often rely on a heuristic which encodes game-specific domain knowledge [10, 29] so that the algorithm can select the most promising actions in the most promising parts of the known gameplay space. While search can be conducted without this information, adding it often dramatically reduces the compute cost required to reach relevant states [10].

We often desire simple and effective heuristics. For games like *Super Mario Bros* [23], where the player's goal is always clear (complete the level by reaching the right side), very simple heuristics like the player's x-position have been highly effective [18, 29]. For highly complex but relatively short games like *Go*,[1] it is practical to learn heuristic functions from vast quantities of data, using both expert gameplay records and self-play experiments [26]. However, automated testing targets a different scenario. For a complex game that is still in the design process, little data is available for statistical modeling, and the specific goals for hand-crafted heuristics are constantly shifting as game mechanics and game world designs change. Seemingly general heuristics (such as encouraging the player to make progress towards an on screen goal by measuring distances in image pixels) are often ineffective in guiding search.

This paper addresses the following research question: **How can imperfect models of a game help in exploration-based testing of that game?** To answer this, we focus on *abstraction guidance*. We use a coarse-grained, incomplete, and even flawed model of the game's mechanics, and show that this is sufficient to usefully guide exploration.

To demonstrate these methods in a realistic setting, we focus on *Super Metroid* [24], an exploration-based platformer published by Nintendo in 1994. We chose this game because it has an active modding community,[2] is easy to interact with via the `gym-retro` Python library [2], and it serves as a compact bundle of many ideas (e.g. game mechanics and world design patterns) present in other videogames [22]. While *Super Metroid* is a platforming game, the player's movement can be influenced by items that they collect; because of this, exploration involves backtracking to previously explored areas with new items. We use a tile-based abstraction described in previous *Metroid*-focused work [22], and we perform search in an emulator for the Super Nintendo console hardware.

[1] *Go* matches typically involve about 200 sequential moves.
[2] https://metroidconstruction.com/

## 2 BACKGROUND

In this section we discuss relevant prior work. Since our method builds directly on Go-Explore, we describe the exploration algorithm in detail.

### 2.1 Search-based Game Testing

Because our target use-case is testing videogames during development, we consider methods for automated playing that can directly execute the game in question.[3] In this case, game-playing can be formalized as finding a path through the state graph of the game. Jaffe explains how many design questions can be posed by setting constraints on the actions allowed during optimal automated gameplay [20]. To convert this path into an *exploration* of the game, existing methods like Reveal-More can be used [4]. To find these paths in the state space, standard graph-based search algorithms like A* could be applied (e.g. [29]). However, for most videogames, search algorithms will only feasibly reach a small part of the game's state space, and finding true shortest paths is infeasible in the long run. Algorithms like Monte Carlo Tree Search (MCTS) add randomness to the search, typically selecting states based on a mixture of the heuristic value and the level of certainty about that value (based on evaluations that start from that state) [3, 21]. While MCTS in more discrete settings can evaluate a state by playing it out until the end of the game (for example AlphaGo [26]), applying MCTS to videogames usually requires some kind of intermediate evaluation (i.e., a heuristic) [18, 29]. For games without obvious heuristics, an existing body of work aims to automatically learn plausible heuristics[12, 15]. However, time spent learning agents for a specific version of the game might be wasted if the next version introduces small changes that obsolete the learned heuristics.

### 2.2 Go-Explore

Go-Explore [10] is another algorithm for automatically playing games. Go-Explore assumes access to a function that can assign any reachable game state into a discrete cell (such that similar states fall into the same cell and meaningfully distant states fall into different cells). This cell-grouping function can be somewhat game-agnostic (e.g. downsampling the screen displayed), or game-specific (including certain details about the state such as global position and even considering state variables not directly visible to the player). During exploration, at each time step, a populated cell is selected randomly, and a known state is sampled randomly from within that cell. Applying random actions to the chosen state produces a new state which is then stored and indexed according to its associated cell. Most implementations of Go-Explore use heuristics to bias random selections of cells, states, and actions while leaving the skeleton of the algorithm in place.

[3] We also assume that the people capable of creating the in-development game are also capable of producing (and maintaining) a simplified model of the game that captures a subset of the original game's mechanics and is capable of working with the original game's world design data. This plausible assumption has not been leveraged in past work on automated testing.

The cell structure prevents the search from focusing too much attention on states that are only trivially different. This helps the search recover from situations where a heuristic has guided the search incorrectly. For example, if the best state (according to the heuristic) is a dead-end, a traditional search can continue to explore nearby trapped states with slight alterations to position and global timer variables. As the system explores these ostensibly valuable states, more will be discovered, and the chance of selecting an actually useful state (with less desirable heuristic score) decreases. In Go-Explore, this cell will still have an incorrectly good heuristic value, but exploring more nearby states will not cause one of these problematic states to be more likely to be selected at each step.

To apply Go-Explore to *Super Metroid*, we use a domain-specific cell-assignment function (described in the next section). Notably, *Super Metroid* and many other games include critical state information that is not always visible onscreen, such as what items the player has collected. This contrasts with the Atari games evaluated in the original Go-Explore paper, and makes defining a good domain-agnostic heuristic very challenging.

## 2.3 Abstractions for Analyzing Games

We rely on the use of an abstraction of *Super Metroid* to generate heuristic guidance. In software verification, abstractions are simplifications of an underlying system that can be analyzed more efficiently than the original system, while retaining key properties [5]. For games, this means a simplified, lighter-weight version of it that does not work identically to the original, but retains some of the key mechanics and a compatible representation of the game's world. The technical games research community often uses abstractions of games in order to apply existing techniques to games where the original version is too complex or cumbersome to use. For example, ANGELINA II, a genetic algorithm for generating mini-metroidvania games, uses an abstraction for playability analysis [7]. Many Mario generation and playing systems use *Infinite Mario* (e.g. [9, 18, 29]) as an abstraction of the original NES game *Super Mario Bros* [23]. Systems have also been built to support creating custom-defined abstractions, such as the Videogame Description Language (VGDL) [25]. The authors describe how approximations of Zelda-like games can be written in VGDL, and note several features that make writing search-based agents easier for these abstractions. Abstractions have also been pre-computed directly from the game mechanics, before applying them to ensure that generated levels are playable [30].

Perhaps the most common type of abstraction used in games today is a navmesh [8]. This decomposes existing world design geometry into a graph, and long-distance navigation tasks are first solved by finding a path in this graph. During gameplay, only the location of the very next navigation node is considered by character control algorithms that move characters on a frame-by-frame basis.

Some game abstractions are *implicit*: they come from design annotation attached to existing world design data. For example, Summerville and Mateas generate playable Mario levels using LSTMs [17]. A key part of their approach is that a playable path for each level is encoded as part of the training data. The model learns about how Mario can move only through these playable paths, so these paths implicitly encode an abstraction of the platforming mechanics in *Super Mario Bros.*

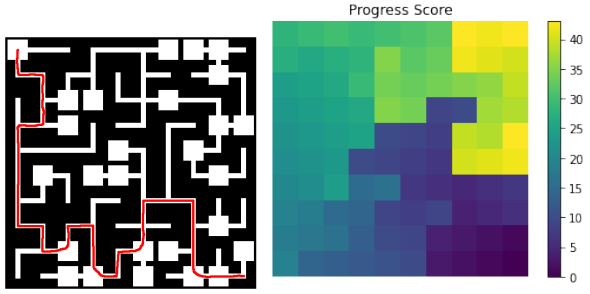


**Figure 2: Left: an example maze with the solution shown in red. Right: the associated progress score computed for the abstraction of that maze. The progress score is the shortest-path graph distance from each node to the goal (lower is closer).**

We use the same abstraction as Mawhorter et. al. [22] which is explicitly designed to model *Super Metroid* at the level of world design tiles (described in more detail in Section 4.1). This prior work searches for paths in the abstraction in order to formally prove properties about the *model* of the games. Earlier work leaves it up to human testers to check that the tile-level counterexamples generated make sense in frame-by-frame execution of the original game.

Although creating such an abstraction is not necessarily easy, the widespread use of similar models suggests it is not unreasonable. In many cases, these abstractions can be created by reading data directly out of the original game's implementation. However, these abstractions will likely ignore features like responsive enemies or destructible terrain elements. In Section 4, we show that Go-Explore can overcome spurious advice from a faulty abstraction and find in-game paths that the abstraction would not have considered possible.

# 3 ABSTRACTION GUIDANCE

In this section, we present a technique for augmenting Go-Explore by providing it with heuristic guidance generated from an abstraction. We illustrate this technique with a toy example.

For a given game, Go-Explore operates over the true state graph of the game, $S = (V_S, E_S)$, consisting of nodes $V_S$ and edges $E_S$. Each node is a state that the player can be in, and each edge is an action that the player can take from that state. We use a separate abstract state graph $A = (V_A, E_A)$, which models $S$, but is much smaller. A mapping function $M : V_S \to V_A$, relates the two state graphs; usually, many real states will map to a single abstract state. We are interested in finding a real path starting at some real state $s_0$ and ending at some real state $s_f$.

The toy example is solving a randomly-generated maze. Each maze is a minimum spanning tree of a randomly-weighted graph. Fig. 2 shows one of the mazes. While traveling through the maze, the player has a 2-dimensional floating point position, with each dimension between 0 and 100, so

$$V_S = \{(x, y) \in \mathbb{F}_{32}^2 \mid 0 \le x, y \le 100, \text{not inside a wall}\}$$

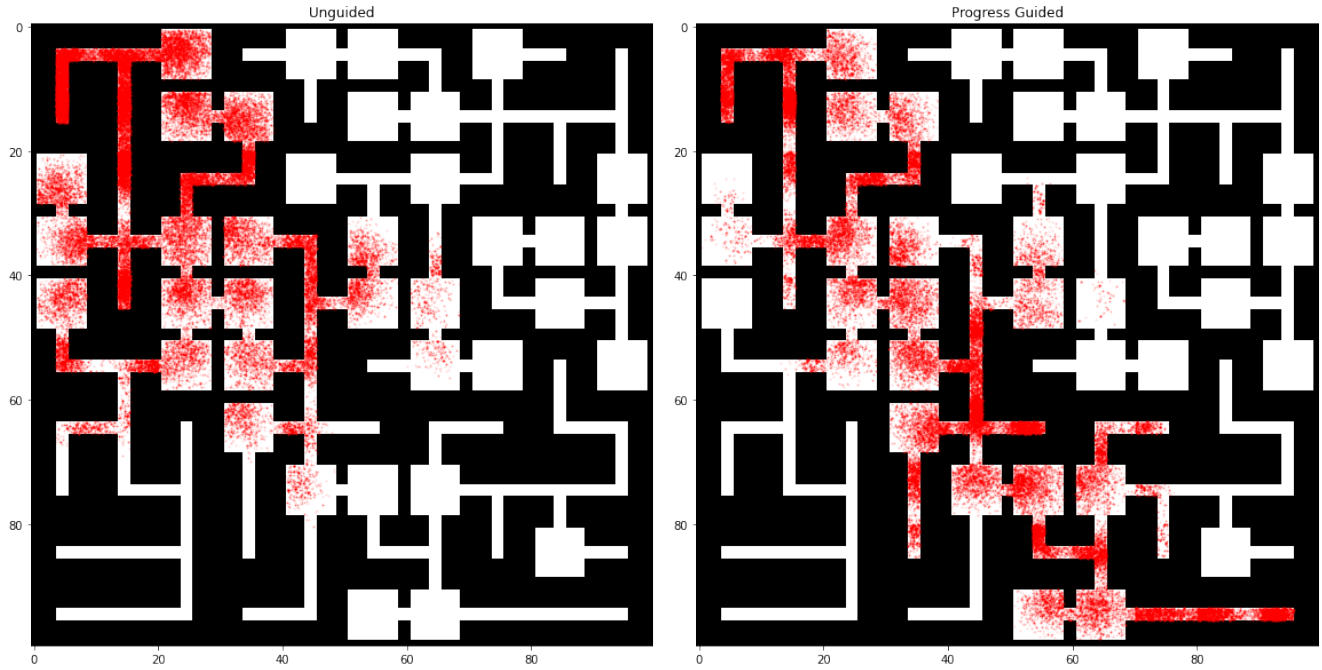


**Figure 3: Results of *continuous*-space exploration of a maze world attempting to find traversal from the top-left of the world to the bottom-right. Unguided exploration (left) explores entire rooms that are irrelevant for making progress towards the goal. Guided exploration with the same total step budget (right) uses *progress* score to bias exploration towards the more relevant cells.**

Where $\mathbb{F}_{32}$ refers to the set of all 32-bit floating point numbers.

In this scenario, the true dynamics are conceptually simple – at each time step, the agent chooses an angle between 0 and $2\pi$, and a distance between 0 and 3 pixels, and moves that far in the chosen direction. A move is invalid if the agent would travel inside a wall. Thus

$$E_S \subseteq \{(a, b) \in V_S \mid ||a - b|| < 3\}$$

The player starts in the top-left room, and their goal is to reach the bottom-right room. Because the agent's position is two floating point coordinates, there are *many* possible states (approximately $1.11e18$)[4], and the entire state graph for even this very simple game cannot be practically generated or stored directly. Although the dynamics are easy to understand, the branching factor is also extremely high, because it involves choosing two floating point numbers. Because of this, using typical graph search algorithms that attempt to consider all possible actions from a given state is infeasible.

Our abstraction of this maze-solving task is a graph of 100 nodes (arranged in a $10 \times 10$ grid), so

$$V_A = \{(x, y) \in \mathbb{Z} \mid 0 \leq x, y \leq 10\}$$

with an edge between two nodes if there is blank space at the border between the corresponding tiles, defining $E_A$. Mapping from the real space to the abstraction is simply done using truncation:

$$M((x, y)) = (\lfloor x/10 \rfloor, \lfloor y/10 \rfloor)$$

[4]https://lemire.me/blog/2017/02/28/how-many-floating-point-numbers-are-in-the-interval-01/

Because the abstraction has a very small graph, a path from start to end can be computed efficiently using breadth-first-search. This results in an abstract path from $M(s)$ to $M(t)$.

The challenge is now to use this path to guide Go-Explore in the real state graph.

## 3.1 Abstraction-Guided Search

To begin, recall that Go-Explore relies on a cell-assignment function, which groups real states into cells. In Mazes, each cell will correspond to an abstract state (a node in the underlying graph), and the mapping $M$ from real to abstract states can be used to assign cells.

When running Go-Explore, the main source of guidance is the *cell selection policy*, which determines which cell will be examined at any given iteration. Typically, this amounts to deciding on a selection probability for each cell, then choosing the next cell randomly according to this distribution.

In order to use the abstraction to guide search, we define the ***progress*** score for an abstract state, which indicates distance to the desired goal. For a given abstract state $a$,

$$\text{progress}(s) = D(a, M(t))$$

This is simply the length of a shortest abstract path from $a$ to the abstract goal. In our examples $D$ counts the number of edges on that path, but in some cases there is a natural weighting scheme associated with each action the player can take. Figure 2 shows what this looks like for an example maze.

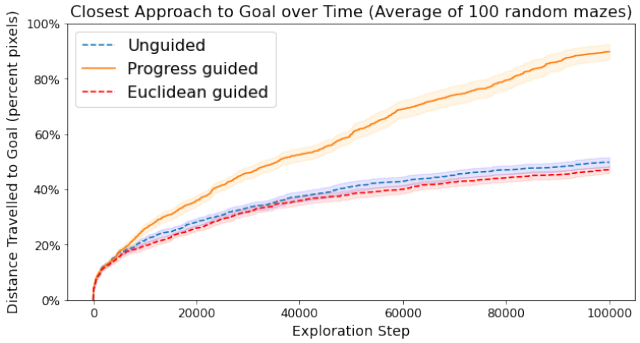


**Figure 4: Progress towards goal over time in 100 randomly-generated mazes, similar to Fig. 3. Shaded region indicates 95% confidence interval, assuming a normal distribution.**

To compute the cell selection probabilities, we use a log-linear model. For the Mazes example:

$$p(s) \propto \exp(-\text{progress}(s)/\beta)$$

Where $\beta$ is a temperature parameter controlling how strong the heuristic guidance is (higher means weaker). In our experiments, $\beta = 3$. A relevant variation (that we do not report on in this paper) reminiscent of Upper Confidence Trees for MCTS [21] would be to decrease the selection probability of a cell proportional to how many times it has already been sampled (subtracting a term proportional to the count in the inner terms of the equation above).

## 3.2 Mazes Discussion

Fig. 4 shows results from adding guidance to Go-Explore on Mazes. Each method explores for 100,000 steps, using the guidance to compute selection probabilities for each cell. Lines show how much of the distance the algorithm has covered over time, with 100% indicating that it has reached the goal. We consider unguided search and two types of guidance. Euclidean guidance chooses cells based on their distance to the goal cell (scaled to [0,1]), while Progress guidance looks at the graph distance from each cell to the goal. The curves were obtained by averaging closest approach on a single run of each algorithm across 100 randomly-generated mazes. Progress guidance solved (got within 1 pixel of the goal) 55 of the 100 mazes within the 100,000-step time limit, while unguided and Euclidean-guided search did not solve any of the mazes.

Notably, Euclidean guidance is about the same as not guiding the search at all (although may be much better for certain mazes). This metric decreases (although not necessarily monotonically) from 1 to 0 in an optimal traversal of the maze, so it might seem better than unguided search. Euclidean-guided search might do so poorly because in this setting, the state space is large enough that the search will never realistically exhaust a dead-end with a good heuristic value. Receiving even a small amount of incorrect advice from the heuristic can dramatically slow down search.

Setting the temperature parameter $\beta$ lower (meaning higher confidence in heuristic values) speeds up progress-guided search, and with $\beta = 1$, progress guidance can reliably solve mazes within 60,000 steps. However, this also means that an imperfect heuristic will slow down the search even more when it gives bad guidance. In Mazes, we know that the heuristic always gives perfect guidance. For applying this technique to *Super Metroid*, we cannot always trust the abstraction.

# 4 SUPER METROID

The Mazes toy example discussed in the previous section is useful for explaining the method of abstraction-guidance and validating some of the results. However, it is not a very difficult problem, and it is easy to see how the abstraction-based heuristic provides extremely good guidance. In order to test our methods in a more realistic setting, we turned to *Super Metroid*, a Super Nintendo game from 1994 [24]. This game has many mechanics that make designing traditional heuristics difficult. For example, the fact that the game requires the player to collect items to proceed implies that strong heuristics will be sensitive to the set of items the player has collected, not just their distance to some waypoint. The inclusion of destructible blocks means that traversibility computed over a static world design is only approximate. The way the game requires the player to backtrack through familiar parts of the map suggests the heuristics will need to provide several different scores for a given point in the world depending on other aspects of the abstracted game state. To cover a few of these concerns, we use the tile-based abstraction described in [22] without changes. This abstraction can be computed automatically for a given part of the game by reading the level design from the ROM and applying a set of defined movement rules to compute reachability.

## 4.1 Abstraction

We performed our experiments in *Super Metroid* using OpenAI's `gym-retro` library to control the SNES9x emulator.

The abstraction we use has a state space consisting of states

$$s = (x, v, i, p)$$

where $x$ is the player character's world-space position (tuple of integers), $v$ is the velocity (tuple of integers with some additional information), $i$ is the item set, and $p$ is the player character's pose (stand, jump, morph, or spinjump). The abstract graph $A = (V_A, E_A)$ is derived from a set of movement rules designed to coarsely approximate the game's physics.[5] In addition to using the same movement model as prior work [22], we also focus on the same section of gameplay, shown in Fig. 1. The abstract graph for this portion of the game has 3,322 states and 13,904 edges. This method for computing the abstract graph scales linearly with the size of the space, and the graph was generated in 10.06 seconds for this example segment of the larger world design.

For a graph of this size, the time spent precomputing the progress scores is still negligible compared to searching within the game. In a more complex example, finding abstract paths might require re-applying a more advanced search algorithm like MCTS or Go-Explore, albeit with a more rudimentary heuristic. Alternatively, the fidelity of the abstraction could be reduced in a way that results in a smaller graph.

[5] Among many other simplifications, the states of all non-player characters are not modeled in this abstraction.

The true state of the game is all random-access memory (RAM) and processor state values for the Super Nintendo Entertainment System (SNES) during gameplay, which consists of 430,181 bytes of data. So, the true state graph for the game is impractically large.

The mapping function $M$ reads the RAM[6] of a given state to evaluate the abstraction parameters, notably truncating the player position value to the nearest 16-pixel tile.

Unlike the abstraction used for Mazes, this abstraction for *Super Metroid* is imperfect. It contains two major categories of error:

- *Overapproximation Error:* It may be possible to reach some abstract states by paths that have no equivalent in the real space. The model has overestimated the player's abilities in some way, and provides incorrect guidance about where to go.
- *Underapproximation Error:* It may be possible to reach some real states along paths considered that have no equivalent in the abstraction. The model has underestimated the player's ability in some way, and cannot provide guidance about states it believes are impossible.

For example, one overapproximation in the abstraction is the treatment of breakable blocks: breakable blocks can be treated both as solid and as air for evaluating the abstract movement model. In reality of course, they are either broken or not in any given state. An underapproximation in the abstraction is the absence of walljumps from the movement model. This causes the model to underestimate the places that the player can reach.

Because of underapproximation errors, the player may sometimes reach states that the abstraction believes cannot reach the goal. For these states, the progress score is $\infty$. However, it may be useful or even necessary to enter these states. For example, the abstraction does not model crouching, but the player character must crouch to complete our examples.

To allow the system to reason about these states, we define an approximate distance function

$$D_{\text{approx}}(s_1, s_2) = \begin{cases} ||x(s_1) - x(s_2)|| & \text{if } i(s_1) = i(s_2) \\ \infty & \text{otherwise} \end{cases}$$

This is the Euclidean distance, but only if the two states have the same item set. When the system discovers a real state where an underapproximation error has occurred, the corresponding node $a$ in the abstraction cannot reach the goal. We first calculate the closest abstract state $a'$ for which $\text{progress}(a) < \infty$ according to this approximate metric. Then define

$$\text{progress}(s) = \text{progress}(s') + D_{approx}(s, s')$$

This allows the system to correctly select crouching states (and other states that are not covered by the abstraction). These approximate values will be increasingly inaccurate as the set of valid abstract states becomes scarcer. Fortunately, the maximum value of $D_{approx}$ seen in our experiments was 2.

To handle overapproximation error, we will rely on the search algorithm to overcome it naturally by eventually finding paths that perform slightly worse than the faulty abstraction suggests should be possible. This makes probabilistic cell selection very important, as *best first* cell selection will not be able to overcome this kind of misguidance.

Finally, in order to practically perform search in *Super Metroid*, we must consider the action set. Naively, the SNES controller has 12 buttons, so the branching factor of search is $2^{12} = 4{,}096$. To reduce this, we recorded one of the authors playing through the level fragment, and restricted the action set to all button combinations pressed during our play (25 total actions that are enough, for example, to express jumping while running in various directions but never using the pause button). The system uses no other information from this playthrough.

Once Go-Explore selects a cell and a state in that cell, it randomly selects an action, consisting of a button combination, and number from 1 to 20 (inclusive). The game is simulated with that button combination pressed for that many frames to obtain the next new state. Because the Super Nintendo nominally runs at 60 frames per second,[7] our average step length of 10 frames corresponds to making decisions about 6 times per virtual second or roughly every 160 virtual milliseconds.

For our experiments, we used progress guidance:

$$p(a) \propto \exp(-(\text{progress}(a))/\beta)$$

## 4.2 Results

The level design in Fig. 1 shows a map of our *Super Metroid* scenarios. The player starts at the bottom of the elevator (A), in the middle of the level segment. Their goal is to get past the missile door on the right side of the screen (E). Since they start with no items, they first need to obtain the missiles (C) to pass through this door. However, blocking the entrance to the missiles is a short passageway that the player cannot walk through (D). To access this passageway, the player must first travel to the left to obtain the morph ball (B) that allows them two squeeze through the small passageway. This example level demonstrates some of the reasons why developing heuristics for *Super Metroid* is difficult. Naively traveling towards the goal will actually make backwards progress (travelling across terrain that then needs to be backtracked). Traveling towards the closest item is better, but some items may be impossible to reach as they require other items to access them, and it may also be necessary to get much further away in order to eventually get closer.

We break this level design up into multiple segments (skipping brief cutscenes manually). Referring to Fig. 1, the segments are A to B, B to C, and D to E. We omit the uninteresting C to D segment. In each of these scenarios, the cell-selection heuristic is the same: path lengths in the abstract graph to the goal at E. The shortest path to E in the abstraction obtains the morph ball in order to obtain the missiles to pass through the missile door at E. This means that the goal of reaching E automatically guides the search from A towards B (and sequentially the other locations) without having to manually specify B as a waypoint. Table 1 shows a comparison between author gameplay and automated gameplay for these scenarios. These values give impression about how long each segment is, how good the paths found by search are, and how efficient the search is.

[6] https://patrickjohnston.org/ASM/Lists/Super%20Metroid/RAM%20map.asm

[7] Our automated gameplay setup, which does not attempt to exploit any form of parallelism, is typically able to simulate around 500 frames per wall-clock second.

| Scenario | Human Gameplay (frames) | Progress-guided path length (frames) | Average States Searched |
|---|---|---|---|
| A to B | 460 | 490 | 1042 |
| B to C | 1240 | 1855 | 9248 |
| D to E | 500 | 603 | N/A |

**Table 1: Expert Human and algorithm gameplay data for the three scenarios. Refer to Fig. 1 to see the scenarios. For D to E, search does not always complete within the time window, so accurate average values are unavailable. Average frames searched is an average of 5 separate runs.**

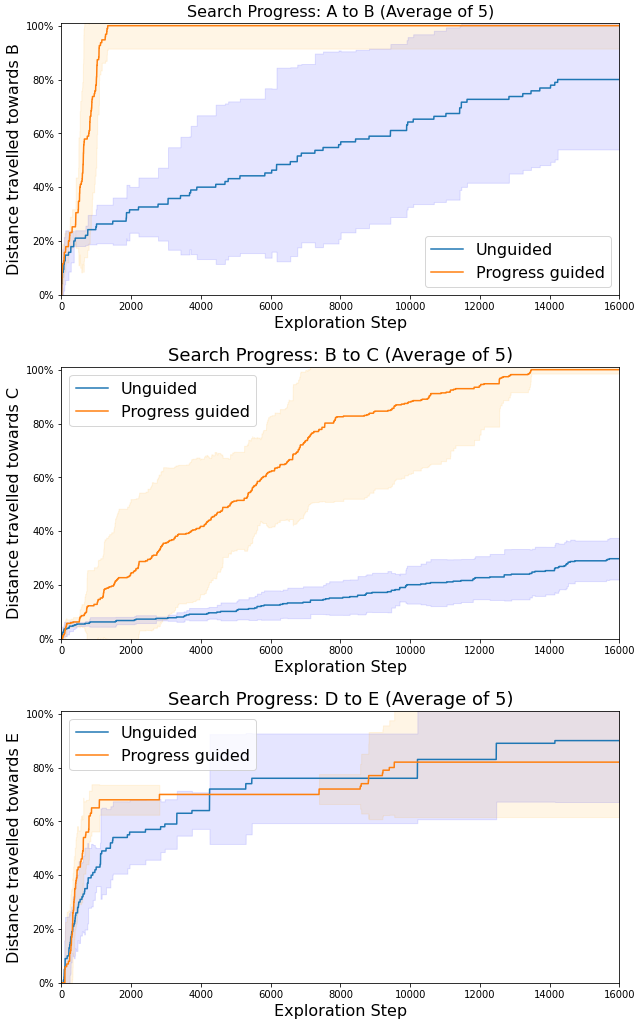


**Figure 5: Summary of exploration progress over time in several gameplay segments from the map seen in Fig. 1. Figures show an average of 5 runs, with 95% confidence interval shown for each run, assuming a normal distribution. Y-axis shows the percentage of the total distance traveled over time.**

Fig. 5 summarizes our results. As with the Mazes example, we measure the distance to the goal over time, averaging over 5 playthroughs. In this case, because Euclidean distance to goal would be misleading, we report abstraction's progress score to show how close the search has gotten to the goal. Progress-guided search vastly outperforms unguided Go-Explore on the A to B and B to C segments. Unguided search was never able to find a path within the time window for these scenarios, while guided search always found a path within 16,000 frames.

The same does not hold for the D to E segment. This segment involves traveling up the room on the right, and to do this the player must climb through the breakable blocks, leaving some in place to use as platforms. Because the abstraction does not model breakable blocks correctly, the abstraction guidance is much less useful during this segment, and unguided search is able to get closer to the goal more quickly than the guided methods. However, this is not the whole story: guided search actually found a solution in two out of five attempts, while unguided search never found a path to the goal within the time cutoff. This suggests that while guided search can still stumble upon a solution more easily than unguided search, it can also be hindered by guidance if it starts out badly.

In addition to mostly outstripping the unguided search, our abstraction guidance results in massive efficiency gains. In the paper introducing the Go-Explore algorithm, the authors report on applying exploration to the simpler Atari game *Montezuma's Revenge* [19], which has some overlapping design elements with *Super Metroid* such as the need to backtrack through the world design as the player collects items. In an experiment they describe, Go-Explore is set to run a simulation of the Atari game console for 1.2 billion frames (or about 230 virtual days of gameplay), producing output trajectories of about 8,000 frames in length[10]. By contrast, by using a stronger heuristic based on guidance tubes, we are able to find useful trajectories about 3,000 frames long with less than 270,000 frames (or 75 virtual minutes) of gameplay. Table 1 reports search efficiency for each segment of our *Super Metroid* scenario.

Once Go-Explore has recovered a path, it can be replayed in the emulator to produce a video as well as a sequence of button inputs that reproduces the path. It can also be used as input to other automated exploration systems like Reveal-More [4].

## 4.3 Limitations

These results show the promise of using imperfect abstractions to develop heuristics that still successfully guide search. However, there are still severe limitations. For example, the missile door on the far right of the screen is still an effectively impossible obstacle. To get through this door, the player must select the missiles, then shoot the door five times in a row. The player has five total missiles, so if any shot is missed, they will be unable to open the door without leaving for an entirely different area to refill their missiles. The

current abstraction is completely unaware of the amount of missiles that the player has, or the remaining health of the door cap, because it models the door as a group of breakable blocks that require missiles.

In testing, the system was never able to discover a way past the door, and this fact is not surprising. Many paths were discovered where the player accidentally fires a missile, creating a state where the player is effectively stuck. Worse, because this does not affect the heuristic in any way, the system will continue to expand these states, and create more states that are stuck, in turn making it even less likely to expand states where opening the door is possible. Even if it manages to get close enough to the door with the right missile count, it needs to shoot the door 5 times, then walk through, a sequence requiring at least 11 consecutive actions. The heuristic does not give any guidance about which of these states are preferable, and the branching factor of the space is 25, almost all of which lead to states that either do not make progress, or become permanently stuck by missing a shot. In short, this system would have to be extremely lucky to open the door without first running out of time or memory.

In this specific case, revising the cell assignment function to include information about the player's missile count is relatively easy. However, it is unrealistic to attempt to resolve all such gaps by making the abstraction more and more complex (approaching the complexity and lower execution speed of the original game). Any kind of feasible abstraction is bound to have some flaws, and not all of them will be easy to recognize or solve.

A more general alternative approach is to run Go-Explore with a finer-grained cell representation and leave the abstraction unchanged. For example, we could make Go-Explore aware of coarse-grained missile accounts. Although this would proportionally grow the state space experienced by Go-Explore, it would have the result of encouraging the algorithm to spread its attention across ways of reaching abstract states with different missile counts (including some where there is enough to pass through the missile door). Because the goal of our project was to find a role for imperfect abstractions (rather than engineering new abstractions or new cell representations), we do not explore the results of this alternative approach here.

## 5 FUTURE WORK

A parallel thread of related research deals with *learning* abstractions using gameplay data [1, 9, 13, 14]. While often this kind of learned model is expected to be used in reinforcement learning, this work shows that search-based methods may also be able to take advantage of this data, or even be used to bootstrap RL agents. Simpler uses of additional data include improving the action selector using player data, similar to the LSTM used by Sorochan et al. [28]. Finally, the learned representation demonstrated by Dann et al. includes probability distributions for action failure [9]. Including this could allow a system to search for the *easiest* path, rather than the shortest path, or even predict the difficultly of paths that players choose to take.

## 6 CONCLUSION

In this paper we considered the problem of making more efficient exploration-based testing systems for videogames. We presented a method for developing sophisticated heuristics based on abstractions of those videogames. We used the abstraction to precompute progress values for the abstract space, then used these values to guide cell selection in Go-Explore. We demonstrated the effectiveness of these heuristics on the motivating example of a simple maze-solving game. We then applied the same technique to quickly find long (3,000+-step) paths in *Super Metroid*, a game for which many naive heuristics will actually hinder search. Overall, our results show how standard automated testing methods to be applied to a broader class of games where the system must make intelligent long-term planning decisions as well as managing movement in the short-term.

## ACKNOWLEDGMENTS

Special thanks to Patrick Johnston for detailed documentation of *Super Metroid*, and help understanding the game internals.

## REFERENCES

[1] Eloi Alonso, Maxim Peter, David Goumard, and Joshua Romoff. 2020. Deep Reinforcement Learning for Navigation in AAA Video Games. *CoRR* abs/2011.04764 (2020). arXiv:2011.04764 https://arxiv.org/abs/2011.04764

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. https://doi.org/10.48550/ARXIV.1606.01540

[3] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[4] K. Chang, B. Aytemiz, and A. M. Smith. 2019. Reveal-More: Amplifying Human Effort in Quality Assurance Testing Using Automated Exploration. In *2019 IEEE Conference on Games (CoG)*. https://doi.org/10.1109/CIG.2019.8848091

[5] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. 2018. *Handbook of Model Checking* (1st ed.). Springer Publishing Company, Incorporated. Section 13.1, p.385.

[6] Jeanne Collins. 1997. Conducting In-house Play Testing. *Gamasutra* (7 July 1997). https://www.gamasutra.com/view/feature/3211/conducting_inhouse_play_testing.php

[7] Michael Cook, Simon Colton, and Jeremy Gow. 2016. The angelina videogame design system—part i. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 2 (2016), 192–203.

[8] Xiao Cui and Hao Shi. 2011. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security* 11, 1 (2011), 125–130.

[9] Michael Dann, Fabio Zambetta, and John Thangarajah. 2017. Real-Time Navigation in Classical Platform Games via Skill Reuse.. In *IJCAI*. 1582–1588.

[10] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. 2019. Go-Explore: a New Approach for Hard-Exploration Problems. *CoRR* abs/1901.10995 (2019). arXiv:1901.10995 http://arxiv.org/abs/1901.10995

[11] Tracy Fullerton. 2014. *Game design workshop: a playcentric approach to creating innovative games*. CRC press.

[12] Cristina Guerrero-Romero and Diego Perez-Liebana. 2021. Map-elites to generate a team of agents that elicits diverse automated gameplay. In *2021 IEEE Conference on Games (CoG)*. IEEE, 1–8.

[13] Matthew Guzdial, Boyang Li, and Mark O Riedl. 2017. Game Engine Learning from Video.. In *IJCAI*. 3707–3713.

[14] David Ha and Jürgen Schmidhuber. 2018. World Models. (2018). https://doi.org/10.5281/ZENODO.1207631

[15] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2018. Automated playtesting with procedural personas through MCTS with evolved heuristics. *IEEE Transactions on Games* 11, 4 (2018), 352–362.

[16] Christoffer Holmgård, Michael Cerny Green, Antonios Liapis, and Julian Togelius. 2019. Automated Playtesting With Procedural Personas Through MCTS With Evolved Heuristics. *IEEE Transactions on Games* 11, 4 (2019), 352–362. https://doi.org/10.1109/TG.2018.2808198

[17] Summerville Adam J. and Mateas Michael. 2016. Super Mario as a String: Platformer Level Generation Via LSTMs. In *Proceedings of the First International*

*Joint Conference of DiGRA and FDG*. Digital Games Research Association and Society for the Advancement of the Science of Digital Games, Dundee, Scotland. http://www.digra.org/wp-content/uploads/digital-library/paper_129.pdf

[18] Emil Juul Jacobsen, Rasmus Greve, and Julian Togelius. 2014. Monte Mario: Platforming with MCTS. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 293–300.

[19] Robert Jaeger. 1984. Montezuma's Revenge. Parker Brothers. Atari 2600.

[20] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popović. 2012. Evaluating Competitive Game Balance with Restricted Play. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Stanford, California, USA) *(AIIDE'12)*. AAAI Press, 26–31.

[21] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. Springer, 282–293.

[22] Ross Mawhorter and Adam Smith. 2021. Softlock Detection for Super Metroid with Computation Tree Logic. In *The 16th International Conference on the Foundations of Digital Games (FDG) 2021*. 1–10.

[23] Shigeru Miyamoto. 1983. Super Mario Bros. Nintendo. Nintendo Entertainment System.

[24] Yoshio Sakamoto. 1994. Super Metroid. Nintendo. Super Nintendo Entertainment System.

[25] Tom Schaul. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. 1–8. https://doi.org/10.1109/CIG.2013.6633610

[26] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[27] Adam M Smith, Mark J Nelson, and Michael Mateas. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 91–98.

[28] Kynan Sorochan, Jerry Chen, Yakun Yu, and Matthew Guzdial. 2021. Generating Lode Runner Levels by Learning Player Paths with LSTMs. In *Proceedings of the 16th International Conference on the Foundations of Digital Games* (Montreal, QC, Canada) *(FDG '21)*. Association for Computing Machinery, New York, NY, USA, Article 53, 7 pages. https://doi.org/10.1145/3472538.3472602

[29] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. 2010. The 2009 Mario AI Competition. In *IEEE Congress on Evolutionary Computation*. IEEE.

[30] Nathan Partlan Vivian Lee and Seth Cooper. 2020. Precomputing Player Movement in Platformers for Level Generation with Reachability Constraints. *Experimental AI in Games* (Oct. 2020). http://www.exag.org/papers/EXAG_2020_paper_13.pdf

[31] Zeping Zhan, Batu Aytemiz, and Adam M. Smith. 2018. Taking the Scenic Route: Automatic Exploration for Videogames. *CoRR* abs/1812.03125 (2018). arXiv:1812.03125 http://arxiv.org/abs/1812.03125