I Can Jump!
Exploring Search Algorithms for Simulating Platformer Players

Jonathan Tremblay and **Alexander Borodovski** and **Clark Verbrugge**
School of Computer Science, McGill University
Montréal, Québec, Canada
jtremblay@cs.mcgill.ca alexander.borodovski@mail.mcgill.ca clump@cs.mcgill.ca

Abstract

Platformer games let players solve real-time, physics-based puzzles by jumping and moving around to reach different goals. Designing levels for this context is a non-trivial task; the placement of well-timed jumps, moving platforms, interesting traps, *etc.*, has a complex relationship to in-game challenge and the existence of possible solutions. In this work, we describe three different search algorithms (A*, MCTS and RRT) that could be used to simulate player behaviour in the platformer domain. We evaluate and compare the three approaches applied to three non-trivial levels, showing a possible iterative workflow of use to designers, and research progress in designing search algorithms for platformer games.

Introduction

The platformer genre is popular amongst indie game developers, mainly due to the easy access enabled by tools like *Unity-2D* or *GameMaker*. Designing meaningful levels for platformers which are challenging and interesting for players is not a trivial task, and it is important to avoid bad designs where players could make use of undesirable shortcuts to solve levels, or which result in overly obscure or difficult solutions. Verifying game solutions, however, is difficult for small-scale developers without the resources to perform large-scale human testing.

We present a study of different search algorithms for finding valid solutions to platformer game levels. The high level goal of this work is to offer the game designer a tool that she can use to see possible solutions to her design. Such a tool can then be included in an iterative design process, and so accelerate and improve development.

Our study has the additional benefit of exploring how different search algorithms can be modified to handle the real-time physics of platformer games, and so be effectively applied to the platformer domain. We demonstrate this through a non-trivial, open-source implementation that offers a tool for designers to apply, explore, and visualize the results of different algorithms in an industry standard game development environment. In terms of academic qualities we identify the following contributions:

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

- Design and presentation of different search algorithms aimed at the platformer domain, and oriented at level testing.
- A comparative study evaluating the effectiveness of the different algorithms on representative platformer levels.
- An open-source implementation in Unity-3D (Borodovski and Tremblay 2014) that enables further extension and exploration.

Background & Related Work

Our focus in this work is in developing different algorithms for simulating player behaviour, as a means of providing tools to better explore the design space of the platformer domain. Platformer games provide a number of unique challenges for such exploration, due to their heavy reliance on physics-based puzzles and their fast action, continuous game-space context. These features are relevant to many 3D-based games, such as *Assassin's Creed*, although here we focus on simpler 2D-based puzzles-games such as *Super Meat Boy* or *Cloudberry Kingdom*, where the player controls a single character who moves through a 2D domain, interacting with the game physics by jumping, running, pushing, *etc.* in order to reach a goal point and avoid various hazards (holes, stompers, *etc.*).

Our work compares the behaviour of different search techniques applied to the design process. In previous work we presented a tool that uses search to simulate possible player paths in stealth-game levels (Tremblay et al. 2013). In that work a randomized search algorithm was used to discover possible player paths, with design visualizations presented based on aggregate heatmaps of the resulting solution sets. Bauer *et al.* also used Rapidly-exploring Random Trees (RRT) for reachable state exploration, although within the context of jumping games using arithmetic parabolas as movement (Bauer, Cooper, and Popović 2013). Sturtevant presented a breadth-first search application to the mobile game *Fling!*, enabling exploration of all possible solutions to a given puzzle set-up (R. Sturtevant 2013).

The first edition of the *Mario AI Competition* (Togelius, Karakovskiy, and Baumgarten 2010) provides a useful benchmark for learning and search techniques applied to the popular series *Mario Bros.*, particularly with respect to player simulation. Baumgarten's controller, for example,

won the competition using an implementation of A*. The solver used a physics simulator to project the resulting position of player actions, a technique we also employ in our work. The output of the solver is then used to figure out which state is the closest to the goal (reaching furthest on the right), and thus which to evaluate next. *Cloudberry Kingdom* by Jordan Fisher uses a similar approach that defines a player AI, which is then used to procedurally generate different platformer levels (Fisher 2012).

Mario AI was not used as a benchmark for our work, as we are interested in building tools for game designers seeking knowledge of their own, new designs, and we did not want to limit expressiveness to the Mario domain. Within this ideology, Nelson proposes different, fundamental questions game designers might ask an AI tool about their designs, such as whether and how something might be possible in the game (Nelson 2011). Our work shows how a tool could answer some of these questions, such as through the use of heat map visualizations that summarize a range of possible game behaviours. Other criteria for AI tools has emphasized the need for game-independent frameworks, that could be reused within different games genres (Smith 2013). In this sense, the general search techniques we examine are highly relevant. RRT in particular has been previously shown to be a powerful search algorithm that can be made game-independent, requiring quite little modification to accommodate specific game mechanics, at least for reasonably simple movement models (Bauer, Cooper, and Popović 2013). Shaker *et al.* presented another, search-based approach to solving continuous-time game constraints in the puzzle-game *Cut the Rope* (Shaker, Shaker, and Togelius 2013). They discretized time, finding the next game state by using a physics simulator to update the world, and then exploring the state space through a depth-first search of the available actions. This allowed them to present a solution to a game level as a path in a tree of game actions. Our search implementations have a similar challenge in discretizing a complex state space, but are applied to larger state spaces, and aim at finding a range of game solutions representative of human exploration.

Other work, such as by Scales *et al.,* lets AI programmers implement their own controllers, using an API within the *Spelunky* GameMaker implementation (Scales 2014). Spelunky is a richer context than Mario Bros, where the player has to survive multiple enemies, break walls to move, gather different resources, *etc.* Scales designed different bots for this context, with different objectives such as gold-digger, or explorer. The controllers developed for this project were based on decision trees, using A* search for simple path planning, but not for actions.

Our work offers novelty in providing an exploration of different search algorithms aimed at solving levels in the platformer domain. Although multiple controllers exist for the platformer domain, we are not aware of other RRT implementations for it using physics simulators, or comparative analyses. In the next section we discuss our overall design, the individual search algorithms, and our (open-source) implementation and visualization strategy.

Algorithm Design

In order to understand and explore the design space of platformer games, we first develop a general state space model that can encompass a variety of platformer games. Different search algorithms can then be applied within this state space in order to discover potential solution paths players may experience. Below we describe the state space, followed by a high-level description of the RRT implementation we use, along with the different motion planners, A* and Monte-Carlo Tree Search (MCTS), that we integrate into RRT and which can also be used as stand-alone solvers. This set of search algorithms is implemented with the purpose of being used offline by game-designers.

State Space

Our representation is aimed at the classical platformer game genre, where the game level is fundamentally a 2-dimensional, Euclidean space. The space is constrained by screen boundaries and physical obstacles, such as shown by the green rectangles in figure 1. A designer may also add to her level various other kinds of obstacles, such as saws, spikes or other kinds of *death* obstacles that kill the player right away, shown as red rectangles in figure 5. Moving platforms are another common feature, and may repeatedly move horizontally or vertically; in figure 6 platform movement is indicated by the gray arrows.

Within the level the player has basic commands to move left-right and jump; we also incorporate double-jumps (allowing a second jump while in mid-air), as a popular, if less physically realistic behaviour. Our physics model includes gravitational acceleration downward, but does not include player momentum on the horizontal axis—a player's forward motion depends entirely on the current left/right key-press, and so may be changed arbitrarily and instantaneously, even while in the air—an approach commonly referred as *air control* by game designers. This gives players fine-grain and highly reactive control, as is common in platformer games.

From this context we can build a formal model of the game state. We define our space Σ as a combination of subspaces:

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}^+_{time} \times \mathbb{R}_{fall} \times \{0,1\}_{jump} \times \{0,1,2\}_{moving}$$

This representation encodes the 2D Euclidean space a player may occupy, a non-negative time vector (essential for representing platform movement), a gravity vector to model the jumping or falling velocity, as well as two discrete domains: a 2-valued domain to indicate whether a double-jump has been performed, and a 3-valued domain to represent motion, as either not moving, moving left or moving right. Note that we are typically interested in the subset of the game space that is not trivially unreachable, and we thus define $\Sigma_{free} = \Sigma - \Sigma_{obs}$, as the space free of obstacles, where Σ_{obs} represents the subset of the space taken by obstacles. We denote a node or state within this space as σ.

We designate the set of player actions defined by the game as A. In our case we have 6 actions, $A = \{\text{jump}, \text{jump-left}, \text{jump-right}, \text{left}, \text{right}, \text{wait}\}$. Actions are

executed in a discrete fashion for the duration t of a single game frame, and applying an action $a \in A$ to a state σ generates a new state, $\sigma' \leftarrow a(\sigma, t)$, where the physics is simulated for the period t—in the case of instantaneous actions, such as jump, we assume that after the action is invoked the entity waits (does not do further actions) for t time. A game also requires starting and goal states, designated as σ_{init} and Σ_{goal} respectively. The starting state is a particular point in our space, $\sigma_{init} \in \Sigma_{free}$, while the goal state is more typically a region, and must also accommodate arbitrary values for the non-location components of the player state. Thus we define the goal as a subset of the space, $\Sigma_{goal} \subseteq \Sigma_{free}$; in our case we use a circular patch of the 2D space of player positions, extruded to a cylinder over the time dimension, and ignoring other components of player state. We represent a player's actual path as an ordered set of actions, separated by a fixed time-step $t' \geq t$. Within this state space, we are interested in finding any path from σ_{init} to Σ_{goal}.

Rapidly-exploring Random Tree RRT is a well known pathing algorithm in the robotics domain, popular for its suitability to high dimensional spaces. In our case, the heuristic and stochastic nature of RRT also allows for a broad exploration of possible solutions, more appropriate for modeling potential player behaviours. Algorithm 1 gives an overview of our implementation which is highly inspired by Lavalle's description (Lavalle, Kuffner, and Jr. 2000).

The algorithm starts by initializing a tree structure in order to keep track of the search process. While a resource (time, tree-size) budget allows, we randomly sample the reachable space and extend the search tree by connecting new points to existing ones. This basic process is complicated, however, by the sparsity of our state space and the need to respect game mechanics. Our approach involves sampling only the basic R^2 space and then constructing the rest of the sampled state based on the previous state of the connecting point. Connections are made by finding the closest node in our tree structure to our sampled state, measured in terms of Euclidean distance $\langle x, y \rangle$ in the plane (line 7). Straight line movements in a platformer are not always possible, and so based on a motion planner (A* or MCTS; see below) we find an actual subpath from the nearest node to the randomly sample node, or as close as we can get—in this way we ensure that the tree grows and that all newly added points are actually reachable. If and once the goal region is reached, a path back to the origin is traced in the tree, giving us a final solution.

Figure 1 shows a visualization of the RRT tree. The edges are indicated by red lines, sampled points by small green dots, and the actual tree nodes added by grey dots. Note that RRT does not generally result in optimal solutions. We consider this beneficial, as such this process allows the game designer to explore the variety of different possible paths players may take to solve the level. With a few changes to the algorithm, however, and enough computation time, it would be possible to show all reachable states, as shown by Morgan *et al.* (Morgan and Branicky 2004). Combined with clustering, this technique creates probabilistic road-maps, such as shown by Bauer and Popović (Bauer and Popović 2012).

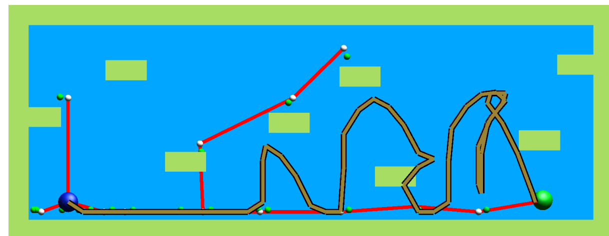

Figure 1: The debug view for the RRT with 20 allowed actions for local search. The large blue and green spheres are the initial and goal positions respectively. Green nodes represent the sampled states, while grey nodes are part of the tree structure, linked by the red segments. The thick brown line shows the path found.

Algorithm 1 RRT applied to platformer domain

```
    procedure RRT(σ_init, Σ_goal, Σ_free, budget)
        i ← 0
        Init(ϒ, σ_init)
        while i < budget do
 5:         i ← i + 1
            σ_rand ← Sample(Σ_free)
            σ_near ← Nearest (σ_rand, ϒ)
            σ_motion ← Motion(σ_near, σ_rand, Σ_free)
            ϒ ← (σ_near, σ_motion)
10:         if σ_rand ∈ Σ_goal then
                return Path(ϒ, σ_motion)
            end if
        end while
    end procedure
```

A* We use A* as both a search process in itself, and as a motion controller in our RRT implementation. Our design is similar to the controller Baumgarten designed for Mario AI, and follows a normal application of A* as describe by Millington (Millington and Funge 2009). Since A*is mainly used to search graph structures, using it in a continuous domain poses some challenges. As described in our RRT design, we use a minimum time step that determines the granularity of the search, assuming each action is applied for a period *t*. Because of this discretization, A*will not ensure an optimal solution.

Figure 2 shows a simplified debug view of the search, with each possible action illustrated by a segment of a different colour: red for jumping, blue for wait, green is left, magenta is right, yellow for jump-left, and white for jump-right (although here jump-right does not appear since both actions jump-right and jump result in the same position). The collision between two states is done using 2 or 3 dimensions $\langle x, y \rangle$ or $\langle x, y, t \rangle$ using simple Euclidean distance, which is also used as our heuristic to reach the goal state. The 2 dimensional approach allows the search to focus on trying to cover as much as possible of the Euclidean space, while the 3D version will explore time as well. This results in a larger state space, but is extremely useful when there is a significant time component to the level, such as the need to interact with a moving platform.

As we will show in our experimentation, the algorithm is fast and travelled distance effective. In the context of a

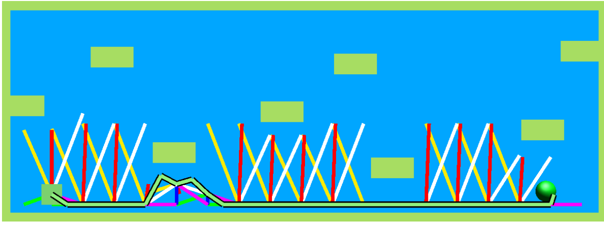

Figure 2: A* debug view. The different colours represent different possible actions at each node, and green square and sphere are respectively the initial and goal positions. The final path is defined by the thick green line.

level-testing tool, however, it is less interesting, as the deterministic nature of the algorithm does not give a designer a good perspective of the breadth of possible solutions that a level design affords.

Monte-Carlo Tree Search (MCTS) Much like A*, MCTS allows us to search a graph. It has a greater focus on graphs with large branching factors, however, with success demonstrated on quite complex games, such as Go (Browne et al. 2012). The strength of MCTS lies in its ability to avoid over-exploring nodes, giving it greater diversity in searching. Algorithm 2 sketches our implementation of MCTS.

In general the algorithm processes states in a way that allows it to focus on a promising branch, deviating to another branch when that may lead to a better result. This process is controlled by the TREEPOLICY method that traverses through the tree, initially invoked on line 6. The first step is to make sure that each available action was applied to the state, and if not, to apply an available action and evaluate the result based on a suitable reward function. The evaluation is back-propagated up the tree, which also increases knowledge about the tree search with data such as the number of times a node was visited.

The MCTS tree search does not take into consideration the proximity to other states as A* does, and naively implemented can sometimes over-search a given location. To avoid this, we added a simple data structure (grid) that keeps track of how many times a grid position was visited. As shown in our REWARD function, we normally rank nodes by the inverse square of their Euclidean distance to the goal, but then also penalize nodes when the player dies or which end up being in over-searched positions (lines 18–20).

Once all available actions have been applied to a state, a child is picked based on its value and the number of times it was visited. For this we simply used the well known upper confidence bounds for tree search (UCT) where the choice of child is treated as the multi-armed bandit problem (Browne et al. 2012). Finally, this process is repeated until the resource budget is exhausted or the goal node is found.

The MCTS search has some stochastic elements, and thus can be invoked multiple times to find different solutions. This leads to a better understanding of the space of possible solutions for game designers, with the benefit that that the search is still very directed, and so the paths found do not wander around the goal.

Algorithm 2 MCTS applied to platformer domain

```
    procedure MCTS-UCT(σ_init, Σ_goal, Σ_free, A, t, budget)
        κ_root ← CreateNode(σ_init)
        i ← 0
        while i < budget do
 5:         i ← i + 1
            κ ← TREEPOLICY(κ_root, A)
            if State(κ) ∈ Σ_goal, t then
                return Path(κ)
            end if
10:         ε ← REWARD((State(κ)))
            PROPAGATE(κ, ε)
        end while
        return null
    end procedure
15:
    procedure REWARD(σ, Σ_goal)
        v ← 1 / Dist(σ, Σ_goal)^2
        if σ ∈ Σ_dead or GridCount(σ) > x then
            v ← −∞
20:     end if
        return v
    end procedure
```

Experiments & Results

The algorithms described in the previous section were implemented as part of an open-source design tool within *Unity-3D* (Borodovski and Tremblay 2014). An important design concern in implementing such a tool in the editor of Unity is the loss of the physics simulator—although it is possible to move *GameObject*s around and check for collisions, the full, built-in physics is not available in editing mode, and thus we were required to implement our own physics simulator for gravity, drag, *etc.* For this we used a discretized simulation with limited collision resolution; for example, while the player is falling, the simulator applies gravity forces to the vertical velocity until the player collides with a platform, at which point movement stops. This limited physics adequately describes many platformer games, and so we leave more advanced collision handling to future work.

Our design allows individual control of the many parameters available to each search algorithm. It is also possible in our design to choose which motion planner the RRT is going to use. Figure 3 shows some of the different parameters available, their impact, and to which search they apply; individual control of these values allowed us to experiment with search behaviour within a wide range of parametrizations.

Performance Tests

To evaluate the results, we implemented different levels that are meant to stress the algorithms in different ways. We first experimented on a few test levels to determine appropriate and practical parameter settings for each algorithm, and then explored algorithm success in terms of searching time, and success ratio, and number of states explored, as well as solution quality in terms of the resulting solution path length, and number of key presses required (as a measure of player effort). For each presented level we ran 1000 searches, using a 16GB Intel-i7 machine, and Unity3D version 4.3.4f1.

Parameters	Effects	Algorithms
Min distance	Minimum distance allowed between nodes	RRT, A*
Max distance	Maximum distance allowed between nodes	RRT
# nodes	Max explored nodes	RRT, A*, MCTS
Time action	Duration of each action	A*, MCTS
2 or 3 dimensions	# of dimensions used in the state collision	A*
Density	Grid size used by the reward function	MCTS

Figure 3: Parameters

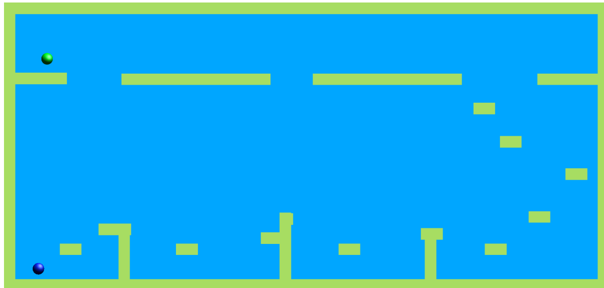

Figure 4: Level 1 where the player has to walk all the way to the right, climb and then move left to the goal.

Numeric results for all tests are given in Table 1.

Level 1 (figure 4) This level is designed to require the player follow a non-trivial path of jumping between platforms, including some amount of vertical ascension.

A* with 2 dimensions performs extremely well, directly heading to the goal, and with search time dramatically less than the others. The A* with 3 dimensions was not able to solve this level as it reached maximum states before reaching the goal; through debug views we were able to see that inclusion of the time dimension resulted in the search failing to make geometric progress as it over-searched the time dimension. The MCTS search took longer than 2D A* by a large factor, reflecting the greater cost inherent in MCTS of having to travel through the tree every time it expands the search.

Using RRT as a higher-level search adds a lot more "noise" to the paths: the average number of keys pressed is significantly higher than A*. This is expected, since RRT does not use any heuristic to bias it search, and so explores more states than the biased searches. RRT_{MCTS} has the greatest diversity in this sense, although it is slow and suffers from a lower success rate; $RRT_{A^\star}$ seems to represent a better trade-off between breadth and performance while still including many random paths.

Level 2 (figure 5) This level provides two options to the player: going straight along the bottom of the level and avoiding the stomper, or climbing up the platforms and using the horizontal moving platform to reach the goal. The biggest challenge with this level is the time component; a player has to either wait for the horizontal platform to traverse the level or time her movement to avoid death by the vertical stomper.

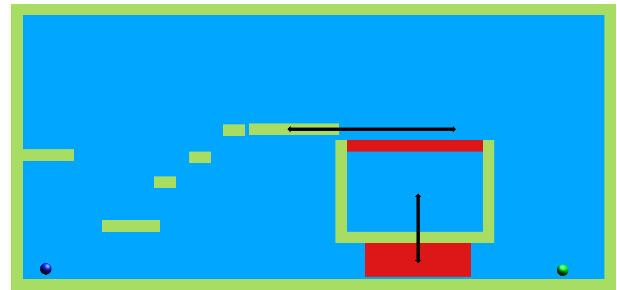

Figure 5: Level 2. Black arrows indicate platform movements, and red regions indicate instant death locations. The top red area can be avoided by riding the platform moving horizontally, and the bottom stomper can be avoided by appropriately timing a dash below it.

In this case both implementations of A* did extremely well. The A* using three dimensions found a marginally faster path, but it was also simpler in terms of keypresses, while the 2D version had to jump around in order to get the timing right. This level is well constructed for A* as the heuristic (Euclidean distance to the goal) leads the search almost directly to the goal. The MCTS search did surprisingly well, given that there is no explicit consideration of time in the search heuristic. The usage of RRT in this level, however, shows a limitation in our sampling heuristic: the tree structure tends to reach a certain time configuration where the node closest to the moving platforms very likely leads to a dying position. Any nodes sampled nearby end up connecting to it and so lead to death, and this is evident in the low success rates for either form of RRT. We expect that a further tuning of the RRT, including a 3D sampling where we consider points in time as well as space, would produce better results, and is part of our future work.

Level 3 (figure 6) This level also gives two options to the player. Going upwards and then right until reaching the end involves few time-based actions (and is the preferred path for both A*s). Alternatively, the player may move to the right and then ride the vertical platform to reach the goal.

The level figure is overlaid with a heat map of the $RRT_{A^\star}$ paths results. As the heat map indicates, most of the RRT solutions follow the same path as the A* (purple path), climbing up and then going across the level. Some, however, either by falling from the top or by moving there from the start do end up using the vertical platform. In the numeric data we can see that A* did well, as the heuristic pushed the search to reach the top right as fast as possible. Both paths given by the two implementations of A* were similar, although the number of states explored is greater by a factor of almost six for the 3D A*. MCTS explores even more states, with slightly lower success rate.

Discussion From these experiments we can see that A* is fast, converging to a solution quickly, and is clearly the

Table 1: Level search results. All times are in ms.

Algorithms	Time all searches	Success rate	Time successful search	Path length in frames	Key presses	States Explored
Level 1						
$A^{\star}{}_2$	392.1 ± 7.3	1.00	392.1 ± 7.3	572.0 ± 0.0	17.0 ± 0.0	785 ± 0
$A^{\star}{}_3$	2251.7 ± 10.8	0.00	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0 ± 0
MCTS	1077.6 ± 246.7	0.97	1062.3 ± 230.1	656.9 ± 44.5	69.5 ± 10.1	4976 ± 1081
$RRT_{A^{\star}}$	2514.7 ± 1441.5	0.96	2354.6 ± 1178.0	759.1 ± 75.8	54.8 ± 9.5	5009 ± 2369
RRT_{MCTS}	13410.9 ± 5921.6	0.84	11703.0 ± 4034.1	1692.9 ± 243.0	109.7 ± 16.7	21179 ± 6488
Level 2						
$A^{\star}{}_2$	228.2 ± 5.1	1.00	228.2 ± 5.1	342.0 ± 0.0	12.0 ± 0.0	542 ± 0
$A^{\star}{}_3$	487.1 ± 6.5	1.00	487.1 ± 6.5	340.0 ± 0.0	8.0 ± 0.0	1246 ± 0
MCTS	897.8 ± 326.9	0.82	791.6 ± 257.8	346.6 ± 33.7	24.3 ± 6.0	4530 ± 1468
$RRT_{A^{\star}}$	1464.6 ± 1621.1	0.67	531.3 ± 800.2	333.4 ± 46.8	7.7 ± 6.4	1230 ± 1763
RRT_{MCTS}	11391.3 ± 5097.1	0.35	8167.1 ± 3873.7	904.7 ± 178.1	47.0 ± 15.4	15889 ± 6935
Level 3						
$A^{\star}{}_2$	1128.0 ± 158.4	1.00	1128.0 ± 158.4	427.0 ± 0.0	23.0 ± 0.0	806 ± 0
$A^{\star}{}_3$	5710.1 ± 447.2	1.00	5710.1 ± 447.2	416.0 ± 0.0	25.0 ± 0.0	4771 ± 0
MCTS	3672.1 ± 952.9	0.95	3593.8 ± 887.1	438.4 ± 34.0	36.9 ± 6.5	5203 ± 1109
$RRT_{A^{\star}}$	3122.4 ± 2563.5	0.69	1792.9 ± 1430.1	516.1 ± 56.9	33.5 ± 6.5	4116 ± 3080
RRT_{MCTS}	16690.3 ± 7365.7	0.49	11261.3 ± 5449.1	1001.9 ± 253.0	64.8 ± 15.6	21502 ± 9723

better approach if a high quality solution is required, which is not surprising. Incorporating the time dimension into the search can be helpful if the level has a strong timing component, but also greatly magnifies the state space and so requires more resources.

From a game-design perspective $A^{\star}$ is useful, but not sufficient—it finds exactly one solution, and so is not helpful in exploring the potential space of solutions in a level. MCTS is then an attractive option as a search process known to be effective in complex domains, and which is also able to find different solutions with multiple searches. From our short study it does not shine as much as $A^{\star}$, being generally slower, but does have the useful ability to show more *human-like*, suboptimal solutions. We note that MCTS also has a drawback in being a less "out of the box" solution, and required some amount of experimentation with parametrization, as well as a customized reward function in order to perform reasonably well.

The random process of RRT shows very interesting results in terms of exploring the space; it tends to find highly variant solutions, and is the most effective algorithm for finding both routes to the goal in our final test. It is, however, by far the slowest, and success can depend strongly on the particular motion planner it uses to connect states. An $A^{\star}$ planner seems perhaps generally better, as it tends to ameliorate the randomness.

Discussion & Conclusion

The presence of non-trivial physics in the platformer domain is a challenge for algorithmic approaches to level analysis. Our experiments show, however, that $A^{\star}$, MCTS, and different RRT designs can still result in effective search processes, whether the goal is to find a single solution ($A^{\star}$) or broad set of solutions (MCTS, RRT) for deeper level exploration.

We are currently investigating different approaches to state sampling in RRT. Improvement should be possible by explicitly sampling the time dimension, although as our 3D

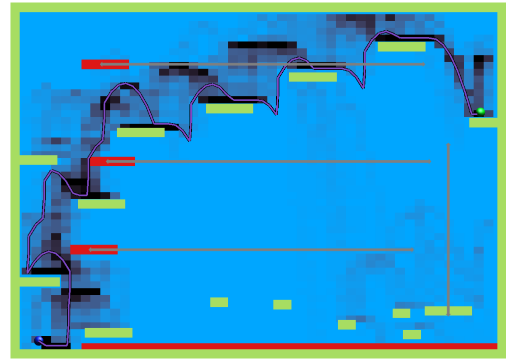

Figure 6: Level 3 with a heap map of $RRT_{A^{\star}}$ results and the $A^{\star}{}_2$ solution shown in purple.

$A^{\star}$ experience shows this represents a trade-off in increasing the search space. Other future work is focused on further analysis of the path quality and variety generated by the different algorithms—total key presses measures player effort to some degree, but speed, length of bursty sequences, and periodicity (Smith et al. 2009) may be better measures of difficulty or appeal. For use in practical level design, we are also interested in how similar the solutions we compute are to human paths. In a greater scheme, we are interested in providing genre-agnostic AI tools for game designers in Unity-3D, useful for both abstract analysis and practical game design.

Acknowledgements

This research was supported by the Fonds de recherche du Québec - Nature et technologies, and the Natural Sciences and Engineering Research Council of Canada.

References

Bauer, A., and Popović, Z. 2012. RRT-Based Game Level Analysis, Visualization, and Visual Refinement. In *AIIDE-2012: Proceedings of the Eight AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*.

Bauer, A.; Cooper, S.; and Popović, Z. 2013. Automated redesign of local playspace properties. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, 190–197.

Borodovski, A., and Tremblay, J. 2014. Implementation for platformer A.I. tool in Unity-3D. https://github.com/GameResearchAtMcGill/unitytool/archive/653c7c.zip.

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Fisher, J. 2012. How to make insane, procedural platformer levels. http://www.gamasutra.com/view/feature/170049/how_to_make_insane_procedural_.php.

Lavalle, S. M.; Kuffner, J. J.; and Jr. 2000. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, 293–308.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition.

Morgan, S., and Branicky, M. 2004. Sampling-based planning for discrete spaces. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, 1938–1945.

Nelson, M. J. 2011. Game metrics without players: Strategies for understanding game artifacts. In *Proceedings of the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process*, 14–18.

R. Sturtevant, N. 2013. An argument for large-scale breadth-first search for game design and content generation via a case study of Fling! In *IDP 2013: Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process*.

Scales, R. D. 2014. Create a set of AI tools for Spelunky that will allow users to program their own player bots. *Discovery, Invention & Application* 1(1). http://commerce3.derby.ac.uk/ojs/index.php/da/article/view/32/27.

Shaker, M.; Shaker, N.; and Togelius, J. 2013. Evolving playable content for Cut the Rope through a simulation-based approach. In *AIIDE-2013: Proceedings of the Ninth AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 72–78.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2D platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 175–182. New York, NY, USA: ACM.

Smith, A. 2013. Open problem: Reusable gameplay trace samplers. In *IDP 2013: Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process*.

Togelius, J.; Karakovskiy, S.; and Baumgarten, R. 2010. The 2009 Mario AI competition. In *IEEE Congress on Evolutionary Computation (CEC)*, 1–8.

Tremblay, J.; Torres, P. A.; Rikovitch, N.; and Verbrugge, C. 2013. An exploration tool for predicting stealthy behaviour. In *IDP 2013: Proceedings of the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process*.