Automated Redesign of Local Playspace Properties

Aaron Bauer, Seth Cooper, Zoran Popović
Center for Game Science, Dept. of Computer Science & Engineering, University of Washington
{awb,scooper,zoran}@cs.washington.edu

ABSTRACT
The design of games can be an unfortunately indirect endeavour: the designer's objective may be to present the player with a particular set of possible experiences, or playspace, but what the designer actually creates (i.e. the game) is instead an artifact intended to produce the desired playspace only upon interaction. In this work we develop both a general framework and specific implementation for providing a designer a way to reason about and edit a game level's playspace. Our system enables a designer to specify edits to playspace properties using a small set of local playspace edits and constraints. It then automatically (via optimization) produces a new level configuration that reflects the desired playspace changes. We present results from our system demonstrating the usefulness of our approach. This work constitutes an initial step toward a fully-realized playspace level design tool.

Categories and Subject Descriptors
I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems – Games; K.8.0 [**Personal Computing**]: General – Games

Keywords
Games, level design, AI-assisted design, covariance matrix adaptation

1. INTRODUCTION
The indirection inherent in game design has expensive consequences. Verifying that a game artifact corresponds to the intended playspace can consume considerable resources in the form of manual testing, redesigning, etc. A tool that enables game designers to analyze and modify their games at the level of play possibilities instead of the literal elements of the game (the location of platforms and numerical parameters) would give them direct access to the space they are trying to design. This, in turn, could make it easier to craft high quality games.

We envision such a tool being employed in an early phase of the level-design process where it could give the designer high-level feedback prior to more costly forms of analysis. It would visualize a level's playspace in a way that would allow the designer to immediately glean useful properties, and would facilitate sophisticated edits to the playspace itself. These edits would be automatically translated into the necessary changes to the underlying level. In this context, we informally define playspace to consist of the the possible sequences of play. This includes the game state at each step and the properties of the transitions between those game states such as difficulty, risk, and importance to the larger space (e.g. a certain transition may be a prerequisite for many states).

This work is a first step towards realizing this kind of tool for levels of 2D platforming games. We use graph-based level analysis from our previous work [1] to generate a graph-based representation of a game level that visually summarizes some properties of the level's playspace. The main contribution of this work is a prototype system that automatically alters a level configuration (e.g. the position of platforms) in order to minimize an objective function over the graph-based playspace summary. The specific edits this work demonstrates are the following: (1) add a transition not present in the original level. (2) Remove a transition that exists in the original level. (3) Adjust the density, or "thickness," of an existing transition, such that is may be "thicker" or "thinner" than before.

As with our previous work, we present the result of this work as applied to a 2D platforming game called Treefrog Treasure, a game developed by the Center for Game Science at the University of Washington[1]. The player controls a frog that sticks to the surfaces of the level (walls and floating platforms). The frog does not run or walk, but traverses levels via a series of parabolic jumps. The objective in each level is to reach a goal location marked by a golden bug. Figure 1 shows Treefrog Treasure in action.

The remainder of this paper is organized as follows. We first discuss related work in automating game design using playspace properties. We then give a brief overview of our previous work in generating graph-based representations. Then, we lay out the details of our playspace edits and present examples demonstrating our results. Finally, we conclude with discussion and future work.

[1] www.centerforgamescience.org/site/games/treefrog

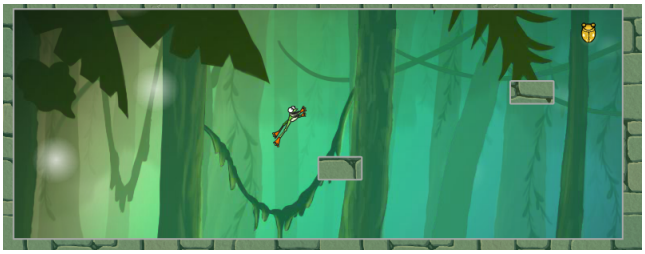

Figure 1: In this simplistic Treefrog Treasure level, the frog (player) begins in the lower left corner and must reach the golden bug in upper right corner to complete the level. The player controls the frog using the mouse, clicking to make it jump.

2. RELATED WORK

Substantial previous work has dealt with automating parts of game design using playspace properties. When focusing on game levels in particular, there are three broad categories.

The first category is systems that generate levels from scratch using playspace properties. There have been a variety of approaches and methodologies in this category. Examples include using a model of challenge-based fun to guide the evolution of 2D platformer game levels [7] and using answer set programming to generate puzzle game levels that obey hard constraint [4]Work in this category has produced very powerful systems for level synthesis, but the lack of interactivity makes them less suited for iterative design than an interactive approach for two reasons. First, if a system can display to the user the possible solutions it considers, it can help the user understand how the system is searching and what possibilities it may have failed to search fully. Second, an interactive system allows the user to reuse their best ideas from the previous cycle in the next iteration.

The second category covers systems that support interactive editing of levels themselves without any notions of playspace. Examples in this category like the editor for the Unity game enginefocus on seamlessly transitioning from editing a level to playing it and back. This is necessary as such context switches are need to verify even small design changes. This approach works to resolve the burden of manual testing without attempting to solve it.

We place our work in the third category: systems that enable interactive editing of playspace. A system called Tanagra allows the generation of 2D platformer levels with respect to user-specified playspace constraints [6]. Tanagra uses a rhythm-based approach, dividing a level into "beats" and letting the designer pin level properties for each beat, such as the position of the platform or the presence of an enemy. The system then automatically shapes the rest of the level to remain traversable from left to right. Our system differs from Tanagra in several ways. First, the graph representation that we base our system on is a more general representation of playspace than the rhythm-based representation used by Tanagra, as it is not particular to platformers or left to right traversal. Due to this, the space of constraints that our framework makes possible is also more general than the constraints Tanagra supports. Our system, however, does not support standard level editing tasks in addition to playspace editing (as Tanagra does), instead focusing solely on editing properties uniquely expressible in playspace.

3. GAME LEVELS AS GRAPHS

The technique described in our previous work [1] for generating graph-based representations of game levels serves as a basis for the tool presented in this work. The details of this method can be found in the original paper. For completeness, we will give a brief overview of the relevant components in this section, as well as highlight the ways in which we changed our previous approach to better to better suit this work.

Like [1], we begin by exploring the level using a probabilistic search algorithm, namely the Rapidly-Exploring Random Tree (RRT) algorithm [3]. In our system, we use the RRT implementation available in the Open Motion Planning Library[8]. This algorithm generates a graph with thousands of states which we then cluster to provide a streamlined representation of the level's playspace.

3.1 Rapidly-Exploring Random Trees

Initially, the algorithm begins with just the player's initial state in the tree. It then iteratively expands the tree, adding new states by taking random actions from existing states already in the tree. Each iteration consists of choosing a goal state uniformly at random from the space of all possible states (i.e. a point anywhere in the level), and then taking a random action from the state in the tree closest to this random goal state. This algorithm requires three game-specific components. (1) A definition of a game state must be provided, since that is what is stored at each node of the tree. (2) To be able to choose the state in the tree "closest" to a random goal state, the algorithm needs a function to compute the "distance" between two game states. (3) To add new states to the tree, a function that, given a current state and an action, produces the resulting state is required.

We define each of these components for Treefrog Treasure. The game statespace is defined as the the frog's position and orientation bounded by the size of the level. The distance between two states is simply the euclidean distance between the frog positions. In our original work, a function to generate successor states was provided by setting the game itself up as an "oracle" where the full game simulation and logic would be employed to compute the next state. This previous approach proved too slow for this work, so we instead compute an analytic solution. Since the frog jumps in parabolic arcs, we simply solve for the intersection of the frog's path with the level geometry. This method is significantly faster and more stable, allowing thousands of simulated actions per second. Figure 2a shows a visualization of a tree for a simple Treefrog level.

3.2 Clustering

To facilitate a cleaner interface for playspace editing, we cluster the original tree produced by the RRT algorithm. The method of clustering is another game-specific component of this pipeline. For this work, we employ a new method of clustering that is both simpler and more effective than the clustering in our original work. Since the Treefrog levels

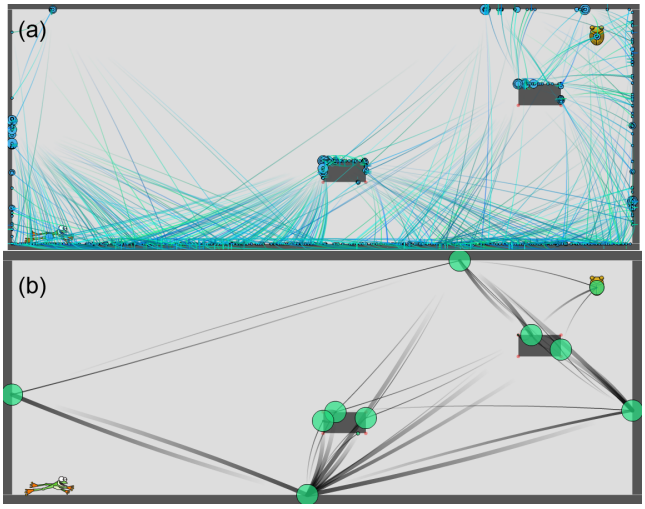

Figure 2: These are visualizations of the tree produced by the RRT algorithm (a) and the corresponding clustering (b) for a simple level from the game Treefrog Treasure. The direction of edges is indicated by transparency. Edges are transparent by their source and opaque by their target. Also, the edges show just the connectivity of the graph, not the actual paths the frog took.

we are considering contain only polygonal shapes, we create a single cluster for each surface that contains at least one node of the tree. A surface is defined as a line segment that constitutes a side of a level element. All the nodes on a particular surface are put into the same cluster. This method of clustering is orders of magnitude faster than our previous method (a few seconds instead of 5-20 minutes), and also results in a cleaner-looking clustered graph. Multiple confusing and redundant clusters no longer appear on the same surface.

The edges in the clustered graph are defined in terms of the edges in the original RRT graph. Specifically, there is an edge from one cluster to another if there is an edge from a node in the first cluster to a node in the second cluster. The weight, or "thickness," of an edge is determined by the number of RRT edges that connect the two clusters.The weight of an edge is related to the precision required for the player to transition from nodes in the source cluster to nodes in the target cluster. The weight corresponds to the frequency of transitions that occurred between nodes in the two clusters during the exploration. Since actions are generated randomly, if only a small number of transitions occurred, then it is likely the target cluster can only be reached from a small set of states, or a relatively precise action is required to reach it, or both. The player may have a harder time accomplishing transitions represented by "thin" edges because the conditions for making the transitions these edges represent are very precise. Figure 2b shows a visualization of a clustered graph for a simple Treefrog level.

4. EDITING PLAYSPACE

Given a graph-based representation of a level, we present a general framework for automated redesign of playspace properties. The core of this framework is a minimization of an objective function, subject to some set of constraints. The minimization is defined as

$$\min_{C} \alpha E_g + \beta E_o$$

where C is a level configuration, E_g and E_o are energy functions for the graph and the elements of the level, respectively, and α and β are weights. E_g, E_o, and C will have particular definitions for different games and/or for different kinds of playspace edits. Constraints are specified over properties of the level configuration C.

To demonstrate the validity of our framework, we have implemented a system that allows for three types of local playspace edits over the edges of a clustered graph. Each type of edit involves a specifying the desired thickness, T, of an edge e in the graph. Our definition of E_g measures how well a candidate level configuration achieves a thickness of T for edge e. The three types of edits are as follows.

- Add a new edge e between clusters where one does not exist in the current clustered graph, also specifying the target thickness T of the new edge.
- Remove an existing edge e between clusters, which corresponds to a target thickness of $T = 0$.
- Adjust the thickness of an existing edge e by specifying a new target thickness T.

As these types of edits all relate to a single edge, they are all local edits to the clustered graph. We chose to focus on local edits in this work because in trying to demonstrate the validity of a new approach, it seemed prudent to focus on the most basic of graph-based playspace edits (i.e. the manipulation of single edges), the understanding of which will hopefully lead to useful implementations of global playspace edits. It is worth noting, however, that our framework lends itself equally well to both local and global edits.

Our system also incorporates constraints, both user-specified and built-in. The user can designate edges to "preserve," during the optimization (i.e. changes to the level that would break the edge are not allowed). There are built-in constraints to prevent level elements from moving outside the boundaries of the level, and prevent platforms being made too narrow (an aesthetic quality). The UI and formalization for these constraints are described below.

The remainder of this section will lay out the details of our system for making playspace edits. First, we'll describe our user interface, and the ways it allows the user to parameterize the subsequent optimization. Next, we'll detail the resampling that is performed to support both the user interface and the evaluation of E_g. Finally, we will discuss the optimization itself, including our definitions of E_g and E_o.

4.1 Level Design UI

Even given the limited set of playspace edits we are considering in this work, there remain many different ways both to specify and parameterize those edits. To this end, we have implemented a small set of naturally visual parameters, but emphasize that our framework places few restrictions on the specifics of the editing interface. The contribution of this work is not the particular UI we have implemented but the underlying system it allows access to.

An edit begins with the user indicating the edge of interest e. This is done in two different ways, depending on the nature of the edit. If the user wants to remove or adjust an existing edge e, they simply click on it to select it. If the user wants to add an edge, they first click on the source cluster, followed by the destination cluster, thus defining (and selecting) a new edge e. After e is selected, the system performs a "dense resampling" of the surface where e starts. The user then chooses a value for T, the desired thickness, in the context of this resampling.

In addition to specifying T, the UI also allows the user to determine the degrees of freedom D. In our current implementation, this consists of selecting (via check box or clicking) which level elements are movable during the optimization. That is, the user decides which platforms are allowed to change as the system tries to find a new level configuration. Each platform is given degrees of freedom based on its type. Rectangular platforms have four degrees of freedom, one for each surface. This means each surface can move along its axis independently. The goal bug is given just two degrees of freedom: its x-y position. The combination of the degrees of freedom of the chosen level elements give us the total degrees of freedom D.

The final part of the level design UI allows the user to put constraints on the optimization by way of designating edges to "preserve." The user clicks a button to enter a "preserve-edges mode," and then clicks on the edges they wish to preserve. The system passes the set of preserved edges P to the optimization.

4.1.1 Dense Resampling

For the purposes of quantifying the current thickness of e as well as evaluating the thickness of e given some change to the level, the node edges from the original RRT graph are not sufficient. For any given e, the number of RRT edges that constitute it may be small (as few as one or two), thus evaluating E_g based solely on those edges is potentially fragile or inaccurate. In our current system, it is computationally infeasible to evaluate E_g by recomputing the entire RRT graph (doing so takes about one minute). Hence, some other source of data about the thickness of e should be used in the context of the optimization.

Our solution is to do a dense resampling of the surface where e starts. This is accomplished by sampling n new player actions where n is 10 times the length of the surface (up to a maximum of 2000 for performance reasons). We define this set of new actions to be A and $a \in A$ to be an individual new action. These actions are sampled from a distribution biased to make it more likely that they hit the surface where e ends. The fraction of $a \in A$ that succeed in reaching the surface of the target cluster is then reported as the quantitative thickness T_e of e and all $a \in A$ are displayed to the user (see Figure 3). This can be expressed mathematically as

$$T_e(C, A) = \frac{\sum_{a \in A} a(C)}{n}$$

where C is a level configuration and given the position of level elements in C

$$a(C) = \begin{cases} 1 & \text{if } a \text{ hits the surface of the target cluster} \\ 0 & \text{if } a \text{ does not} \end{cases}$$

The user then chooses a value for the desired thickness T, since they can now choose that value in context (i.e. choose it relative to T_e). A is passed along to the optimization, and is used to recompute T_e for candidate level configurations.

The concept of dense resampling is not game-specific. We perform dense resampling on a surface because in our implementation on Treefrog Treasure, each cluster corresponds to a surface. If applied to a different game, the system might need an alternative definition of the subset of game states relevant to a cluster. In any case, the general concept still stands that a dense resampling should be conducted on the space of game states associated with the cluster where e starts.

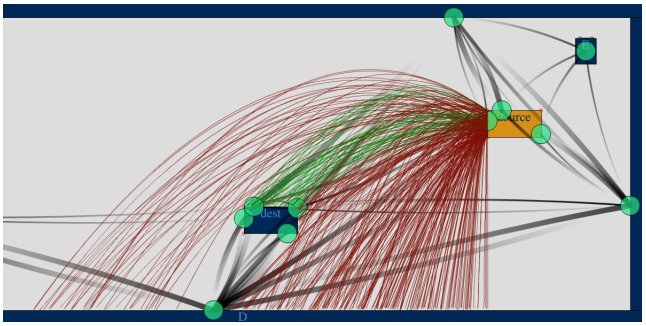

Figure 3: A dense resampling for a clustered edge chosen by the user. The red and green lines show the path of the frog as a result of each sampled action. The green lines are those that hit the target surface and the red lines are those that did not. The platforms are blue with labels to make them distinguishable when the user is determining the degrees of freedom. When a platform is designated as movable, it turns yellow.

4.2 Optimization

Our optimization takes as input the current level configuration C, the desired thickness T, the actions from the dense resampling A, the preserved edges P, the degrees of freedom D, and the current type of edit, whether its remove, add, or adjust. It returns to the user a new level configuration C'.

4.2.1 Algorithm

To perform the optimization and generate C', we use Covariance Matrix Adaptation Evolution Strategy (CMA-ES). Originally described in [2], CMA-ES is an evolutionary algorithm that uses covariance matrix adaptation to update the covariance matrix for the distribution it uses when sampling

new candidate solutions. CMA-ES has several nice properties that make it appropriate for our system. (1) The search space of level configurations is quite rugged and CMA-ES is well-suited for rugged search landscapes. (2) CMA-ES requires little in the way of parameter tuning, which is convenient since we would like a single optimization algorithm for multiple types of edits. (3) Since we are only considering transformations to level elements (as opposed to adding or removing elements entirely), our searches are over a fixed number of dimensions, which is appropriate for CMA-ES.

4.2.2 Representation

We represent a candidate solution c (a candidate solution is a representation within CMA-ES, different from a candidate level configuration) as a set of floats, where each c_i is the change in position for one of the degrees of the freedom, $d_i \in D$, selected by the user. For example, if the user set one platform and the goal bug to be movable, then each c would consist of six floats; the first four would be the changes in position for each of the four sides of the platform, and the last two would be the changes in x and y position of the goal bug. The deltas $c_i \in c$ are applied to the corresponding level elements in C to produce C', which is then evaluated by the fitness function to assign a fitness value to c.

4.2.3 Constraints

Given a set of preserved clustered edges P, we define p to be the set of RRT graph edges that constitute these preserved edges and $p_i \in p$ be the individual RRT graph edges. Since each RRT graph edge is associated with a particular player action, we can evaluate $p_i(C)$ in the same way as $a(C)$. Thus, the constraint imposed on the optimization by preserved edges is: for each $p_i \in p$, $p_i(C) = 1$. That is, no changes to the level are allowed that would cause any RRT graph edge that is part of one of the preserved clustered edges to no longer hit its target surface.

The built-in constraints are as follows. (1) We prevent platforms from being too narrow. Let k be a platform, and w_k and h_k be the width and height, respectively, of that platform. Then, for all k, $w_k > 10$ and $h_k > 10$ (enforcing a minimum size of 10 pixels). (2) We don't allow level elements to move outside the boundaries of the level. Let l be a level element and l_x and l_y be the x- and y-positions of l. Also, let b_x and b_y be the maximum x- and y-values allowed by the level boundaries. Then, for all l, $0 < l_x < b_x$ and $0 < l_y < b_y$.

Since our optimization algorithm is not capable of enforcing hard constraints, we convert these hard constraints into soft constraints that are evaluated as penalties assigned to the fitness function. We set the values of the penalties such that they overwhelm the other components of the fitness function. The exception is the penalty for platforms that are too thin, which is less severe since it a mostly aesthetic concern. The penalty for platforms outside the level boundaries is assessed in a binary manner (either it is applied or it isn't). The penalties for not preserving edges and thin platforms are assessed incrementally.

4.2.4 Fitness

The fitness function follows the minimization described above. E_g is defined to be how close C' is to achieving a thickness of T for e. E_o is defined to be how different C' is from C.

Specifically, the fitness of a level configuration C' produced by applying c to C is expressed as

$$f(C', c) = \alpha E_g + \beta E_o + penalties$$

$$f(C', c) = \alpha \left(T_e(C', A) - T\right)^2 + \beta(\sum_{i=0}^{|D|} c_i^2) + penalties$$

where *penalties* encompasses the the penalties for violating the three constraints described above. Recall that $T_e(C', A)$ is the thickness of edge e for level configuration C' given a dense resampling A. Note that the second term (the sum over c_i) is the sum of the squared deltas for each degree of freedom in D. Hence, the optimization attempts to achieve the desired thickness while changing the level as little as possible (and avoiding violating the constraints).

We set α and β such that the optimization prioritizes matching the target thickness over minimizing changes to the level (i.e. the first term has more effect than the second term). We also use different α and β for different tasks, namely using a larger α for remove edits than for add or adjust edits. This is because a remove edit does not accomplish its goal unless there are no remaining connections whatsoever, whereas the result of an add or adjust edit is acceptable if it gets close to the target thickness.

4.3 Recomputation

Once the optimization is complete and a new level configuration has been generated, the last remaining task is to integrate the existing graph with the new configuration. To accomplish this we use the same local recomputation technique described in the previous work [1] with a few modifications. Given the knowledge of which elements of the level were changed, the process is as follows. First, we identify the set of directly affected clusters, which is simply the clusters located on those elements that were changed. Next, we recompute the results of all the RRT edges coming into and going out of nodes in the affected clusters, updating the graph as necessary. We then remove any nodes that no longer have any incoming edges edge, in case they have become unreachable due to the changes.

As a final step, we compute new actions sampled from nodes throughout the level in order to discover new connections that have been made possible by the changes. The probability that a new action is sampled from a particular node is proportional to that node's proximity to changed level elements. After the action is computed, the new action and state are only added to the graph if together they introduce a new edge in the clustered graph. This prevents the recomputation from vastly increasing the number of states in the graph (which is harmful to both performance and the visualization) and is acceptable because representing the correct reachability via the graph is more important than ensuring the edges of the graph are precisely the correct weight.

5. RESULTS

As with our previous work, we implemented our interactive system on top of an existing level editor for Treefrog Treasure. The level editor, built in ActionScript, displays a

schematic view (i.e. monochromatic platforms and walls) of the level currently being edited. Our graph for the level is overlaid on top, physically placing it in the context of the level. Using the level design UI described above, the user specifies an edit and its parameters. This information is then sent to a Python server, which runs CMA-ES. The server sends back the best candidate solution from each generation (once every 1-2 seconds), which the level editor displays, giving the user feedback on how the search is progressing. The optimization to produce the new level configurations took between 40 seconds to four minutes, depending on the number of degrees of freedom and the number of actions in the dense resampling. In this section we present four example uses of our system, and sketch out how those examples fit into a level design scenario.

The example images have been annotated to make clear what is happening. There is unfortunately not space here to include images of all stages of the UI, so annotations allow for both brevity and clarity. All platforms in the level are marked with blue rectangles to help them stand out. Platforms designated as movable for the optimization are surrounded by a yellow outline. The chosen edge being edited is highlighted in red, and any edges designated as preserved are highlighted in purple. The images within each figure show the state of the level before and after each edit, proceeding chronologically from top to bottom.

In Figure 4, the designer has created a level where the player must jump through a narrow gap to get from the left side of the level to the right side. The edge connecting the left and right platforms looks too thin, however, so the designer sets it to be thicker. To prevent the optimization from making the left platform unreachable, the designer designates the edge from the floor to the top of the left platform to be preserved. Lastly, the designer sets the left and right platforms to be movable.

Figure 5 starts with a level where the designer has laid out three platforms with the intention of each providing a route to the goal bug in the center. Unfortunately, only the middle platform is close enough, requiring two add edits in a row. The designer adds an edge first from the top of the left platform to the goal, and then from the top of the right platform, designating existing connections between other platforms and the goal to be preserved. The designer also designates the connection from the ground to the top of the middle platform as preserved, since, unlike the left and right platforms, it is not reachable from the walls. At each step, the designer allows the goal, the platform they wish to connect to the goal and all platforms currently connected to the goal to move.

For Figure 6, the designer has a level with three ways to get to the goal, but they don't like the one from the low, middle platform. They want to remove it, but also preserve the paths to the goal they like, so they perform a remove edit on the offending edge while also designating the other paths to the goal to be preserved. Since the designer doesn't have a firm idea of which platforms to move to best accomplish this, they set all the platforms to be movable.

Figure 7 shows a sequence of three adjust edits. In the original level, all the platforms are the same size, and about the same distance apart. The designer decides that this is boring, and that it needs some spicing up. To make this happen, the designer first sets the thickness of the edge between the top of the first and the top of the second platform to be thinner, allowing just the second platform to move. The designer then repeats this pattern with the following two edges, making the edge from the second platform to third platform thicker and the edge from the third platform to the fourth platform thinner, always only allowing the destination platform to move.

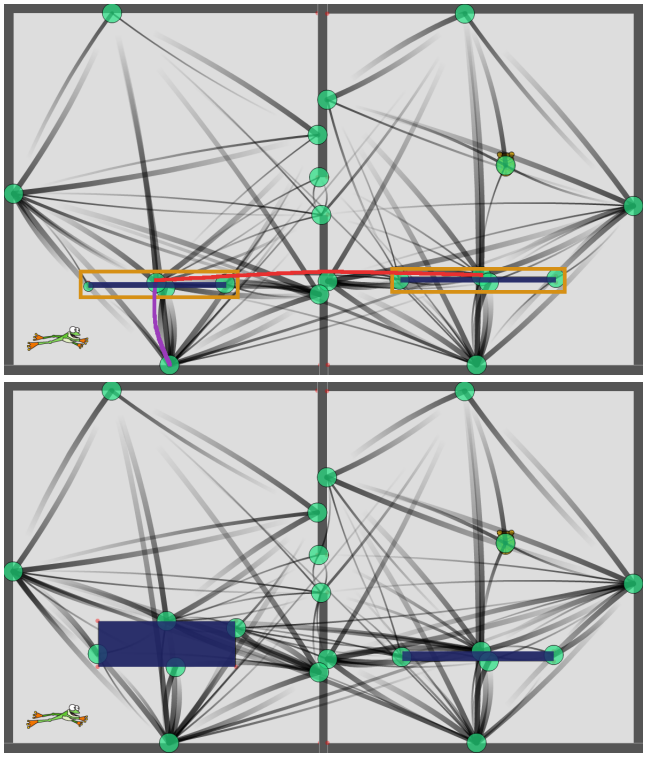

Figure 4: A simple adjust edit where the edge is thickened. The system moves the top of the left platform up as much as possible without severing the edge from the floor, which thickens the edge from it to the right platform considerably.

6. DISCUSSION

We believe we have demonstrated the potential of our system to provide game designers with a way to edit local properties of a level's playspace. The framework that we present here is very general. Any constraint that can be specified over a graph representation and then evaluated as part of an objective function can be incorporated into our system. To this point, we have also demonstrated that a graph-based representation of a level can function as an effective representation of playspace in the context of a playspace editing system. This, along with the generality of generating graph-based representations as explained in our previous work, suggests our approach could be applied to a variety of games.

The implementation presented in this work has several limitations. (1) The only types of level elements we consider are

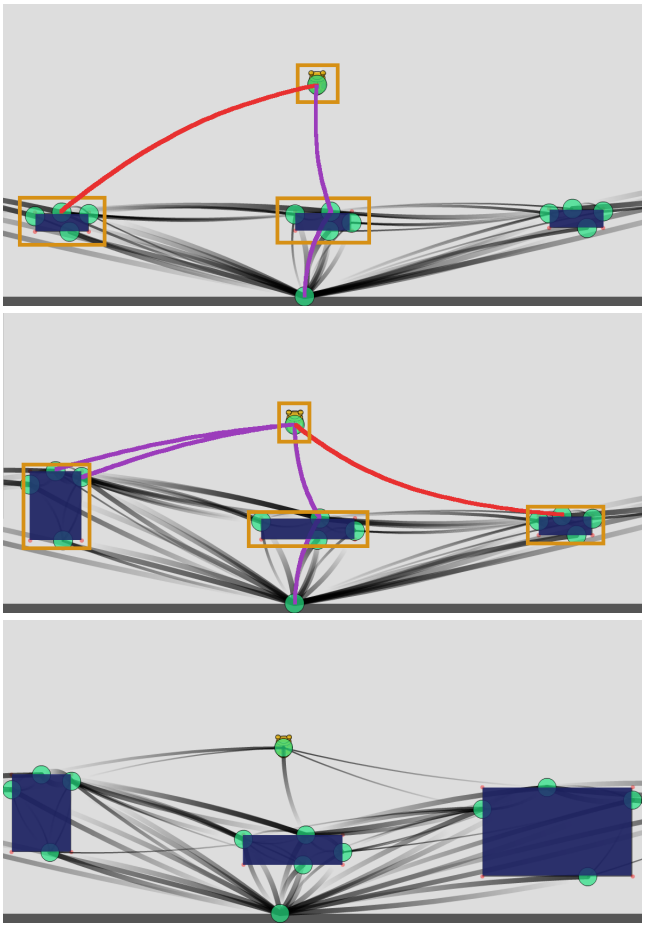

Figure 5: A series of two add edits with the results arranged chronologically from top to bottom. The final configuration allows the goal to be reached from each of the three platforms. As a result, the left and right platforms are no longer reachable from the ground, but fortunately they remain reachable from the adjacent walls (not visible in these images).

rectangles and the goal bug, which is modeled as a rectangle of fixed size. We chose to do this to simplify our implementation in both the computation of player actions and the dimensionality of the optimization. It is not, however, a requirement, and our system could certainly be extended to accommodate other types of level elements. (2) We don't support modifying size of level or level boundaries in any way as part of the optimization. We view these modifications as global changes, and this work focuses on local changes. Also, modifying the level boundaries can result in an ill-formed level with holes that a player can jump through and leave the level entirely, and we chose not to incorporate a check for this into the fitness function. (3) We don't allow the user to add an edge from an existing cluster to a surface that has no cluster (i.e. that is not currently reachable in any way). Again, this is not a fundamental limitation in any way, but would require the system to be able to "snap" a user's target location to a valid position actually on the intended surface, and no such functionality is implemented at this time.

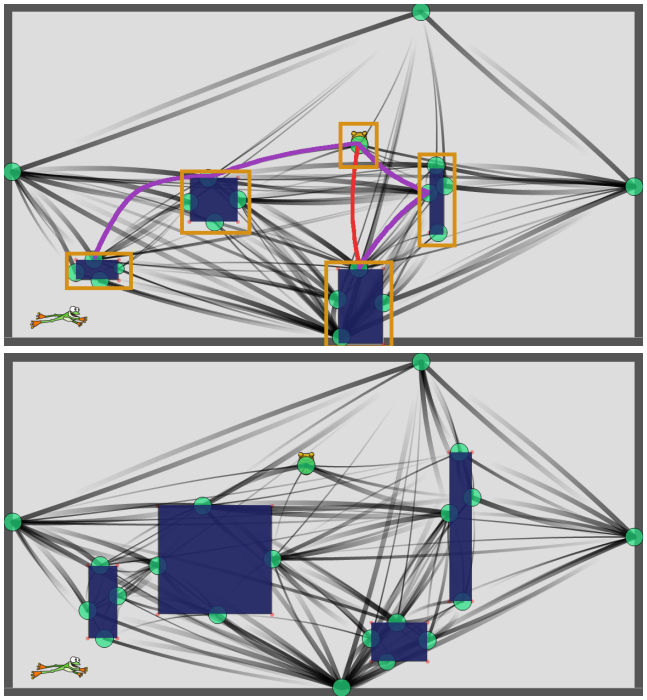

Figure 6: A remove edit in which all the platforms in the level were allowed to move. In the final configuration, the level has been reshaped in an interesting way that also satisfies all the constraints.

These kind of limitations are acceptable, as the purpose of this work is to demonstrate playspace editing and this can be done on a subset of game features. We adhere to the general principle that it is useful to show how our approach supports additional automated intelligence in the design process even given the simplifications we have made [5].

It is the case that our system does not produce good results in all cases, a fact that should be unsurprising, particularly for a prototype system like ours. Sometimes a new level configuration might be uninteresting, or unplayable, or simply not satisfy the specified constraints. This can be the fault of the user, if the constraints or degrees of freedom are poorly chosen, but it can also be the result of the optimization finding some undesirable way of meeting the constraints. For example, Figure 8 shows the result of the same task as Figure 4, but where all four platforms were allowed to move. The system satisfied the constraints, but in a way that doesn't make sense in the context of the full game mechanics. Further refinement of the fitness function could eliminate some of these cases.

7. CONCLUSIONS AND FUTURE WORK

The framework and system presented here are initial steps towards realizing a tool that enables many kinds of sophisticated edits to a level's playspace. A great deal of future work remains. Applying our system to an additional game, preferably one with notable differences from Treefrog Treasure, would reinforce the generality of our work. It would also be exciting to expand the optimization to support a search space of varying dimensions. This would allow it

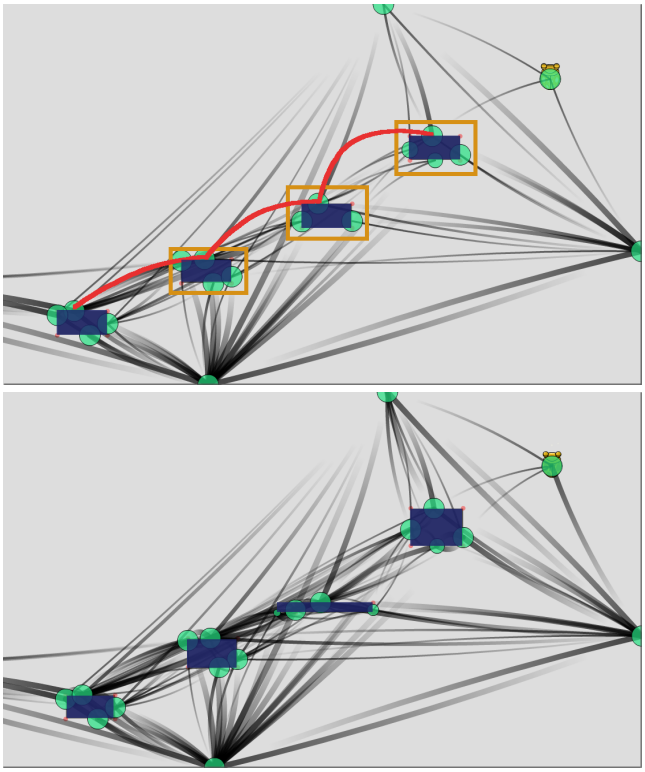

Figure 7: A series of three adjust edits. To conserve space, the annotations for all three edits are shown together in the top image and the cumulative result is shown in the bottom image. After the three edits, the designer has a level with varied platforms presenting a series of varied jumps. Automating this kind of "add variety" series of edits as a single operation applied to a path is an exciting direction for future work.

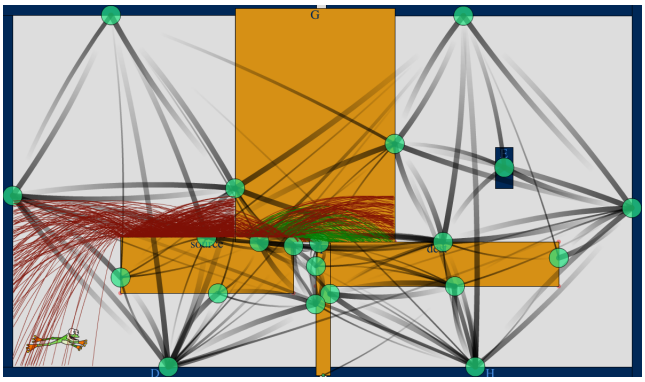

Figure 8: The same edit as Figure 4, except all four platforms were allowed to move. The system achieves the target thickness by placing both the start and target surfaces inside a blocking platform. This is possible because our analytic computation of the frog's parabolic jumps only checks for intersection, not the type of empty space the frog travels through (i.e. inside versus outside a level element).

to consider level modifications that alter the dimensionality of the search, such as adding or removing platforms. This might be accomplished by simply running a fixed dimension algorithm like CMA-ES multiple times, but more likely will require a new algorithm such as Reversible Jump Markov chain Monte Carlo.

An obvious and promising avenue for future work is the addition of global playspace edits or interesting combinations of local playspace edits. This could include edits such as the ability to ensure a level has at least three distinct paths to completion all of approximately the same thickness, or ensure a level's distinct paths to completion are sufficiently different. As mentioned in Figure 7, local edits could be batched together as a single unit to accomplish a task such as making an entire path thinner or thicker.

Despite the limits of our current implementation, we believe it demonstrates the validity of our general framework for editing playspace properties and suggests intriguing directions for future work in this space.

8. ACKNOWLEDGEMENTS

We would like to thank the creators of Treefrog Treasure: Yun-En Liu, Tim Pavlik, Seth Cooper, Marianne Lee, Barbara Krug, François Boucher-Genesse, Fernando Labarthe, Eleanor O'Rourke, Erik Andersen, Brian Britigan, and Atanas Kirilov. This work was supported by the University of Washington Center for Game Science, DARPA grant FA8750-11-2-0102, and the Bill and Melinda Gates Foundation.

9. REFERENCES

[1] A. Bauer and Z. Popović. RRT-based game level analysis, visualization, and visual refinement. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[2] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. In *Proceedings of the IEEE Conference on Evolutionary Computation*, 1996.

[3] S. M. LaValle and J. J. Kuffner. Randomized Kinodynamic Planning. *The International Journal of Robotics Research*, 2001.

[4] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the Conference on Foundations of Digital Games*, 2012.

[5] A. M. Smith and M. Mateas. Computational caricatures: Probing the game design process with ai. In *Proceedings of the Workshop on Artificial Intelligence in the Game Design Process*, 2011.

[6] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[7] N. Sorenson and P. Pasquier. The Evolution of Fun : Automatic Level Design through Challenge Modeling. In *Proceedings of the Conference on Computational Creativity*, 2010.

[8] I. A. Sucan, M. Moll, and L. E. Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 2012.