RRT-Based Game Level Analysis, Visualization, and Visual Refinement

Aaron Bauer and **Zoran Popović**
Computer Science & Engineering
University of Washington
Seattle, WA 98195, USA

Abstract

Automating parts of game creation benefits both professional and amature game designers and much previous work has already made progress on this front. In this paper we tackle automating level design. We describe a general graph-based representation for game levels and present a preliminary system that leverages this representation. Our system automatically explores existing levels of a 2D platform game using the rapidly-exploring random tree (RRT) algorithm and constructs a compact graph representation from this exploration. Our system can also modify a graph representation on-the-fly to reflect user-directed changes to the existing level structure. This work constitutes an initial step toward the larger goal of automating level design in a general way.

Introduction

Many components of game creation have already received varying levels of automation and even optimization in previous work. In terms of automatic game creation, we envision a system similar to the one described in (Togelius and Schmidhuber 2008) where, given a space of game mechanics, the system would optimize over possible games. The goal would be an instantiation of a set of mechanics from that space possessing certain desirable properties (e.g. good balance, appropriate difficulty, etc.). A system like this, together with other content generation tools or collections of pre-authored content, could significantly increase the accessibility of digital game creation. An important part of such a system, and a valuable tool in its own right, would be a technique for automating the analysis and refinement of game levels.

Fully analyzing a game level requires considering a number of questions. Perhaps the most basic among these is reachability; whether the player can reach a given state. The designer may want to ensure the level's goal is actually attainable, or ensure a challenging part of the level can't be circumvented. Beyond this there are deeper questions one might ask. For example, "how many distinct paths can the player take to reach the goal", "how does the difficulty of these path compare", and "how difficult is the level overall" all seek a descriptive characterization of the player's experience. A designer might ask meta-questions such as "what makes this level difficult," or similarly, "where is the player most likely to have difficulty." We believe our approach can begin to answer questions like these, and our primary contribution lies in efforts to answer the latter meta-questions. In this paper, we develop a compact graph representation for game levels, and then visualize this representation to support level analysis and design.

The remainder of the paper is organized as follows. We first discuss previous work in automating both game and level design. We then lay out the details of our graph representation and our approach to constructing instances of it. Then, we describe our implementation and provide example use cases. Finally, we conclude with discussion and future work.

Related Work

There have been a number of efforts to automate the design of game rules or mechanics themselves. Work in automatically generating board game rulesets includes (Browne and Maire 2010). The authors' Ludi system used evolutionary search to discover new games and evaluated them through self-play simulations. The quality of new games was assessed according to 57 aesthetic criteria of the self-play results, ranging from duration to drama to uncertainly. Self-play policies were guided by 20 different "advisers" for game aspects such as mobility, proximity to goal, attacking potential, etc. The relevant advisers for each ruleset were automatically derived and then the policy optimized through evolutionary search.

As for generating digital games, Togelius and Schmidhuber (2008) automatically generated and tested single-player Pac-Man-like games using volutionary computation. The fitness function used to evaluate evolved rulesets focused on the new games' "learnability." To measure this, neural networks were evolved as AI players for the new games, with the fitness of a given ruleset was based on how well the AI players performed.

Other work has sought to automatically generate, analyze, and test game levels. In (Sorenson and Pasquier 2010), levels for 2D platform games are generated to maximize player fun. The authors maximize player fun by modeling the challenge presented by jumps between platforms and generating levels that maintain an appropriate level of challenge throughout. Notions of player anxiety, periodic challenge and "flow" are also incorporated. The authors' implementation uses a genetic algorithm to generate levels for *Super Mario Bros*.

BIPED, the system presented in (Smith, Nelson, and Mateas 2009), provides a game sketching language in which game designers can use to prototype their game in a way similar to how video games are often physically prototyped. BIPED enables designers to use machine testing to investigate questions about the game that would be tedious or impossible to answer with human players. Similarly, our framework allows a designer to visualize a level's reachability, a property that requires an exhaustive manual search, and is thus ill-suited for human testers. Smith has substantial other work on automating game creation (Smith and Mateas 2010) and level design (Smith et al. 2012).

Darken (2007) has much in common with this paper. Darken uses an autonomous agent to create a waypoint graph for levels in a 3D first-person environment. Like our approach, the exploration is done using in-game actions and simulation, and the resulting graph can be used to assess a level's reachability. Darken forsees the waypoint graph being used by AI-controlled characters in-game and so, in a second phase, the nodes in the waypoint graph are annotated for viewshed, cover, and visibility.

Turning Game Levels into Graphs

One of our goals for our level representation is for it to be as general as possible and a graph is well-suited to this purpose for several reasons. First, the structure of a graph makes the general formulation independent of any specific game. More concretely, the nodes of the graph are game states reachable by the player in the context of a level, while the edges denote the possible transitions between these states. For the representation to remain general, the process of turning a level into a graph must also remain agnostic as to the mechanics of any specific game. Hence, in our general framework, we deliberately avoid using a game's logic directly in gathering a level's reachable states. Given this constraint, for any game that isn't simple to the point of triviality, it will likely be arduous, or even intractable, to enumerate all reachable states. Our solution is to use a probabilistic search algorithm, namely the Rapidly-Exploring Random Tree (RRT) algorithm (LaValle and Kuffner 2001), to sample a level's state space. This produces a tree with hundreds of game states as the nodes. We then condense this into a more descriptive graph by applying a graph clustering algorithm called the Markov Cluster Algorithm (MCL) (van Dongen 2000). The resulting graph forms a reasonable model of the level under consideration.

Rapidly-Exploring Random Trees

In our system, the RRT algorithm functions as follows. It begins with only the player's initial state, s_0. Each iteration i it selects a "goal" state, g_i, uniformly at random. It then finds the state currently in the tree, s_i, closest to g_i, and executes from s_i a player action selected uniformly at random, a_i. The new state, s_j, resulting from this action is then added to the tree, T. Let s be a game state, a be a player action, and $\textsc{Move}(s,\ a)$ be a function that returns a state s' that is the result when a is applied at state s. Then we say that for iteration i, add s_j to T where $s_j = \textsc{Move}(s_i,\ a_i)$. If applicable, the state corresponding with the player's actual goal, g, can be given to the algorithm, along with a goal bias probability p_g, and g will be selected as g_i with probability p_g. The reason the algorithm is "rapidly-exploring" is because it explores from the node closest to g_i. Intuitively, the largest regions of unexplored space are adjacent to the "frontier" of the tree, and this is where the algorithm is most likely to explore from.

Clearly, as the RRT algorithm requires a notion of distance between game states and a way to simulate player actions, it must use game logic in some way. This logic, however, is incorporated in a modular way, meaning the basic algorithm is unaffected. Specifically, there are three components that are dependent on the game in question. First, a definition of a game state, $\mathcal{D}$ must be provided to establish what data is associated with each s_i. $\mathcal{D}$ will be a vector indicating the domain for each component of the game state. For example, if, for a given game, the game state tracks one of three modes and a 2D position, then $\mathcal{D} = \big(\{A,\ B,\ C\},\ \mathbb{R}^2\big)$. Second, a function $\textsc{Dist}(s_1,\ s_2)$ from pairs of game states to real numbers is needed to compute a distance metric for states. This is necessary to allow for the notion of a "nearest" state. Third, the function $\textsc{Move}(s,\ a)$ is required to expand the tree. The tree T output by the algorithm can be written in terms of these components. For each $s_i \in T$ except the initial state s_0, we can write it as (s_{i-1}, a_{i-1}), its predecessor and the action that explored it. Hence, the tree is created by the connectivity between generated states. See Figure 1a for a visualization of the tree for a 2D platform game level.

Clustering

Unfortunately, the tree output by the RRT algorithm is not particularly useful to us in isolation. It might contain hundreds of states and transitions, and simply not provide the high-level, compact representation we require. We address this by using MCL to cluster the tree. The input to this algorithm is a list of pairs of nodes along with the similarity between the two nodes in each pair. Here, again, a modular, game-specific component is required. Namely, a function that computes the similarity between nodes in the tree. The output of MCL is a partitioning of the nodes into clusters. More formally, we say the output of MCL is a clustering C, containing n mutually-disjoint clusters $\{S_0,\ \ldots,\ S_n\}$. Each cluster $S_i = \{s_i,\ s_j,\ s_k,\ \ldots\}$ is some subset of the nodes from the original tree T.

Once we have C, we can construct an informative graph representation of the level. The clusters become the nodes in the graph. The edges between the clusters are created and weighted according to the edges in T; there is an edge between two clusters if there are edges between the nodes in those clusters. Formally, there exists an edge e_{ij} from S_i to S_j if there exists some $s_k \in S_j$ such that $s_{k-1} \in S_i$. The weight of e_{ij} is equal to the number of such $s_k \in S_j$ that exist. This weight is a natural metric to apply to inter-cluster edges, as the frequency of transitions from S_i to S_j may correlate with the difficulty for the player of making the same transition. If only small fraction of states in S_i have a transition to a state in S_j, it suggests either S_j can only be reached from a very small set of states, or S_j can only be reached by very specific actions, or both. We assume that a task will be more difficult for a player if they must start from precisely the correct location or execute precisely the correct action. In this way, from a clustering C, we construct a graph G that gives a designer some basis on which to answer questions of reachability and difficulty. See Figure 1b for a visualization of the graph for a 2D platform game level.

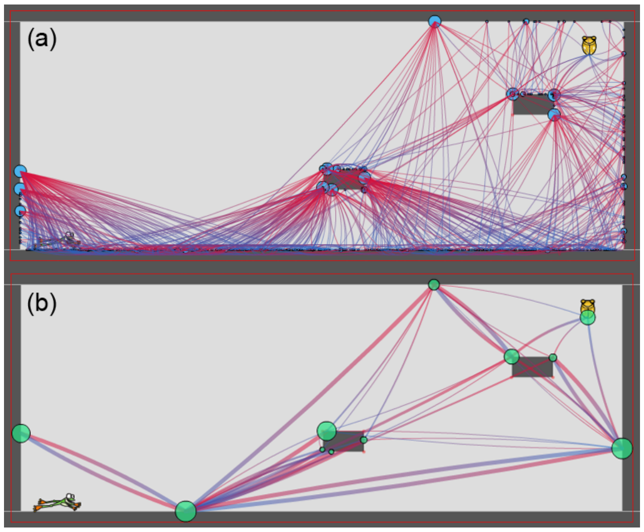

Figure 1: These are visualizations of the tree produced by the RRT algorithm (a) and the corresponding clustering (b) for a simple level from the game Treefrog Treasure. We describe out application to Treefrog Treasure in detail below.

Graph Recomputation

Though a static graph is a good start, it does a poor job of supporting rapid, iterative refinement of a level. An additional layer is needed to enable the designer to quickly investigate how changing the level affects reachability and difficulty. Our approach is to introduce a system that recomputes the graph in real time in response to changes to the level. In essence, we present the designer with an interface that displays both the level and the graph G, overlaying them if it semantically relevant to do so (e.g. the level's state space includes a geometric coordinate space, so it make sense to physically place in the level). Then, when the designer makes changes to the level, G is recomputed to reflect the altered level and displayed accordingly. In this way, the designer is able to get immediate feedback on the high-level impacts of the changes they are making. We envision this system being built on top of an existing level editor for the game in question. As a level editor is often a vital tool in its own right, we believe this is a reasonable prerequisite.

To determine the exact recomputation to perform we employ the following process. First, after a change is made to the level, we identify the set of directly affected clusters, C_a. This identification can be done via proximity or by using some more sophisticated heuristic. After clusters are identified, for all $S_i \in C_a$, we compute $\text{MOVE}(s_i,\ a_i) = s'_i$ and $\text{MOVE}(s_{i-1},\ a_{i-1}) = s'_{i-1}$ for all $s_i \in S_i$ using the altered level (we recompute all incoming and outgoing edges for all nodes in all affected clusters). We adjust G to reflect where s'_i and s'_{i-1} differ from the results computed using the original level. In addition, we compute the results of random actions from a small number of states distributed throughout the level to increase the chances of finding undiscovered graph edges introduced by the designer's change. For this system, we focus on changes that affect the level locally, rather than changes that affect the level as a whole. This system is intended to be employed after an initial level design is complete to support analysis and refinement of the design. The key is that since we restrict the scope of changes to be local, we are able to focus our recomputation of the graph appropriately. This increases both the speed and accuracy of the recomputation because it means fewer samples needed overall and allows us to concentrate sampling where the level actually changed.

Application to Treefrog Treasure

To put our graph representation for levels into action, we implemented our the system described above for a 2D platform game called Treefrog Treasure being developed by the Center for Game Science (CGS) at the University of Washington. The game has a large variety of features, but for this initial implementation we chose to work with a limited subset.

In our limited version, the player takes on the role of a frog that sticks to the surfaces of a level, which consist of walls and floating platforms. The frog begins a level at a specified location, and the objective is to reach a goal location marked by a golden bug. The player interacts using the mouse, clicking to make the frog jump. The position of the mouse controls the direction and speed of the jump; the farther away the mouse is from the frog, the greater the speed of the jump. Once the frog is in the air, the player can take no action until it lands. Gravity affects the frog while it is in the air. Figure 2 shows Treefrog Treasure in action.

We implemented the necessary game-specific components for Treefrog Treasure: the definition of state space (simply the frog's position and orientation bounded by the size of the level), a euclidean distance function over the frog's position, and a function to compute the outcome of player actions. The latter was accomplished by setting up the game itself as an "oracle" that could, given a starting state and a player action, produce the resulting state. This required minor instrumentation of the game's code to allow it to execute arbitrary actions from arbitrary states. We used the RRT algorithm implementation available in the Open Motion Planning Library (The Open Motion Planning Library (OMPL) 2010). Treefrog Treasure is implemented in ActionScript and OMPL in C++, so we had Treefrog Treasure act as a web server to which the RRT algorithm could connect and submit its state-action queries.

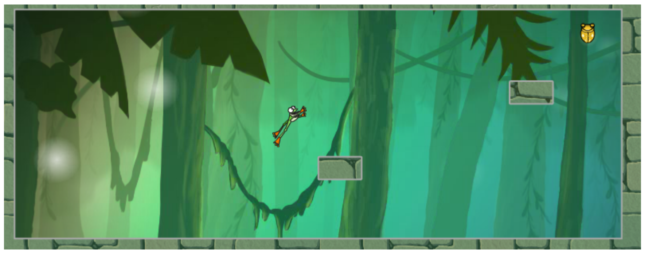
Figure 2: In this level the frog begins in the lower left corner and must reach the golden bug in the upper right in order to complete the level.

To perform the clustering we used the implementation of MCL available at `micans.org/mcl`. The similarity metric we used for clustering is based on the proximity of nodes' predecessors and successors. We first used the XML file specifying each Treefrog Treasure level to extract the surfaces in the level. We only calculated similarity for nodes on the same surface; nodes not on the same surface implicitly had no similarity. To calculate the similarity between nodes a and b, we find the predecessor of a and the predecessor of b that are closest together. Let these nodes be a_p and b_p, respectively. We do likewise with successors of a and b to get a_n and b_n. We then compute the distance d_p between a_p and b_p and the distance d_n between a_n and b_n. The similarity of a and b is inversely proportional to the quantity $d_p + d_n$.

Visualization

Visualization of the graph was implemented on top of the Treefrog Treasure level editor, also built in ActionScript. The level editor displays a schematic view (i.e. monochromatic platforms and walls) of the level currently being edited. It allows a designer to manipulate the elements of a level in various ways, including specifying translation, scaling and rotation for existing elements, removing existing elements from the level, and adding new elements to the level. Our graph for the level is overlaid on top, physically placing it in the context of the level. The clusters are displayed as circles with size proportional to the number of constituent nodes. The clusters are located at the same location as their highest-degree member node. The edges are displayed as curved lines, and are colored with a red-blue gradient to indicate their direction; red by their source and blue by their destination. Figures 3, 4, and 5 show the visualization applied to several Treefrog Treasure levels.

Recomputation

Within the Treefrog Treasure level editor, when the user changes an element in the level, we identify the affected clusters by proximity to the changed element. Specifically, any clusters located on a surface of the changed element are considered affected. As described above we then recompute the incoming and outgoing edges of the nodes in those clusters. In addition, we compute the results of a new action from several random nodes in each cluster. We bias these actions to have high speed and to be in the direction of the

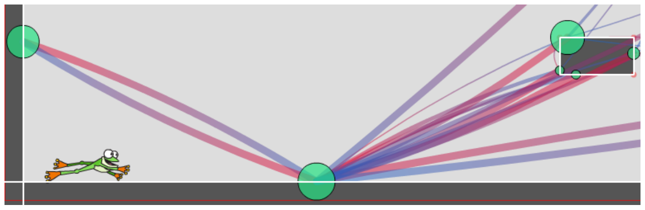
Figure 3: This is a portion of a graph near the starting position (indicated by the frog) in a Treefrog Treasure level. The dark grey objects are the walls and platforms. Note how there is a single cluster for each surface. A couple important observations are made possible by the visualization. First, by the thick edges, we see that it's probably easy for the player to reach the left wall and the bottom and sides of the platform from the floor and vice versa. Second, it appears considerably more difficult for the player to reach the top of the platform, indicated by the fact thin blue edges connect to the cluster there.

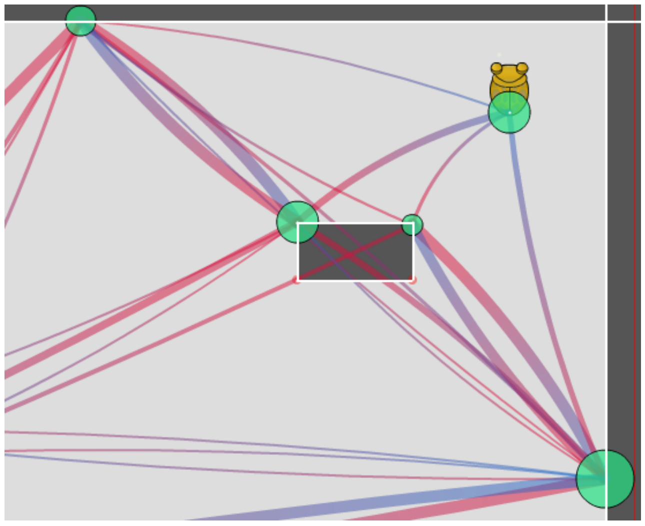
Figure 4: This is a portion of a graph near the goal position in a Treefrog Treasure level. We can immediately see that there are four edges leading into the golden bug marking the goal. A closer look reveals, however, only the cluster on the right wall has incoming edges from clusters other than the five clusters visible here. This indicates that the easiest path of the player is first to the right wall, then to the goal or the right side of the platform.

changed element. Figure 6 shows a visualization of this recomputation. In order to achieve high enough performance to provide real-time interaction, however, we approximate the recomputation. Instead of using the full Treefrog Treasure engine to recompute each edge, we treat the frog as a point and compute the parabolic path this point follows given the speed and direction of the action. Though this approximation does not take into account the area of the frog, it is still accurate enough for the purpose of visual refinement. Figure 7 demonstrates a use case of our recomputation.

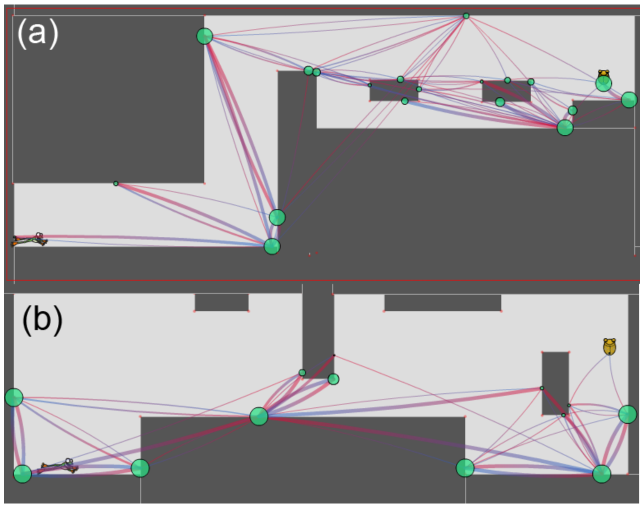

Figure 5: Here are two other Treefrog Treasure levels with their corresponding graph representations. In (a) we can see the player has only one path available initially, but as they near the goal, the graph branches out, indicating more player choice. In (b) we note it appears difficult or impossible for the player to reach the goal from above; the only edges to it in the graph come from below.

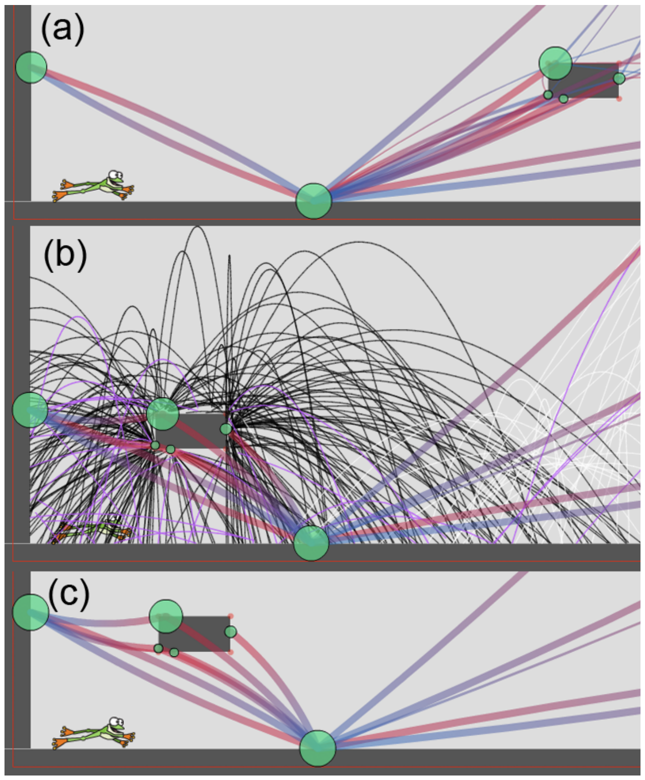

Figure 6: At the top, (a) shows the initial level configuration. The user then moves the platform to the left, so the graph needs to be locally recomputed. When recomputation occurs, the user has the option to view a visualization showing all the individual recomputed actions (b). The black lines are outgoing edges that resulted in different states than in the original tree. The white lines are the same, but for incoming edges (we see that they pass through where the platform used to be). The purple lines are the new random actions. (c) shows the new graph following the recomputation.

Discussion

The first thing to note is the degree to which our approach enables a level designer's to analyze and refine a level. We have shown the usefulness of both our visualization and our recomputation. As detailed in the figures, our system can help a designer identify what areas of a level a player can reach, and how those areas are reached. In addition, a designer can then get real-time feedback as they edit the level. It is also worth noting that our more general framework for RRT-based level analysis exists independent of a particular game or genre and could serve as the foundation for varied applications. One important aspect of our framework that remains largely hpothetical, however, is the correlation between edge weight and difficulty. We are assuming that in most cases a transition that requires higher precision is more difficult, but we have yet to validate this assumption with player data.

Our application to Treefrog Treasure, however, is an incomplete realization of our framework. First of all, it currently functions on only a small subset of the Treefrog Treasure game. It does not yet support dynamic objects within a level other than the player and it relies on an approximation for the graph recomputation. In general, the recomputation should be able to reproduce every action. For the purposes of this application, however, we chose to sacrifice a small amount of accuracy in order to achieve interactive speeds. We believe it is possible to implement the recomputation such that it is entirely accurate and provides the necessary performance, but we leave this for future work. Furthermore, our recomputation does not currently support all the operations available in the Treefrog Treasure level editor. Rotation is not supported, and modifying the four outer walls that form the boundaries of the level can cause instabilities such as supurious nodes. We plan to support all level editor operations in the future.

Though our graph recomputation is done at interactive speeds, our current implementation achieves real-time feedback only imperfectly. After the user specifies a change to a level element, all the recomputation is done as a batch, resulting in about a second of lag before the graph updates. Exactly how long an update takes depends on the number of edges that have to be recomputed, with most updates involving several hundred edges. To accommodate the workflow of a level designer, the recomputation would need to be done on-line; the system would respond immediately and then gradually refine the graph as the results of the recomputation streamed in. The implementation presented here, however, is sufficient for a proof-of-concept that such recomputation is both feasible and potentially useful.

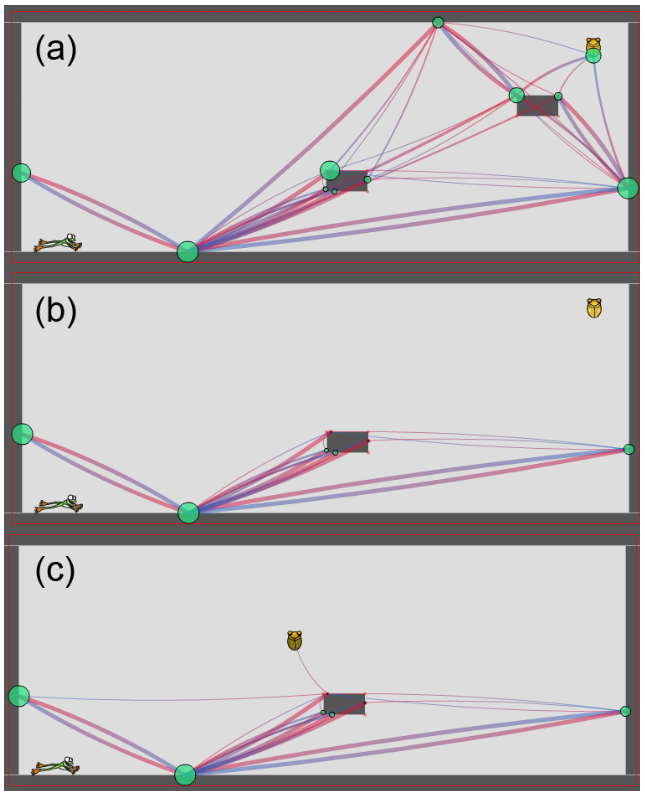

Figure 7: This shows a use case for our interactive graph recomputation. At the top, (a) shows the initial level, where the designer sees that the player can reach the goal by jumping from the right wall to the upper right platform. The designer decides they would prefer that this wasn't possible and deletes the offending platform. The recomputation kicks in, showing the designer (b), alerting them to the fact that the goal is no longer reachable. The designer then moves the goal to above the remaining platform and the recomputed graph confirms it is now reachable (c).

Conclusions and Future Work

Our framework for representing levels as graphs along with our application of that framework to Treefrog Treasure serve only as initial steps in establishing a general approach to level analysis, visualization and refinement. As such, there remains a great deal of future work. Our general framework could be expanded to automatically address a far greater number of useful queries about game levels. One such addition would be the automatic detection of paths in the graph from the start state to the goal state. These could be highlighted for a designer along with relevant features, such as the most difficult edge in each path or any "bottleneck" nodes that every path must pass through. This would enable the system to not only show a designer the effects of the changes they make, but to highlight areas where changes might be needed.

Another, more ambitious, extension would be to tackle the "inversion" of out current recomputation. Specifically, we would allow the user to manipulate the graph, and then perform optimization over the space of local changes to the level to produce a solution level that was the best fit for the modified graph. In other words, in addition to turning levels into graphs, we would also turn graphs into levels. If this technique were to be developed further, perhaps an entire level could be synthesized from a graph, enabling a designer to make compact, high-level changes to generate a whole suite of related levels.

The framework we've presented here provides a way to reason about the complexity of game levels and could form the basis for many future automated tasks. In a way, the graph representation we describe is an abstract language for expressing the challenge presented by an individual game level. We believe it offers a rich avenue for further research with a number of possible applications.

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.

References

Browne, C., and Maire, F. 2010. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1):1–16.

Darken, C. J. 2007. Level Annotation and Test by Autonomous Exploration. In *Proceedings of the Third International Conference on Artificial Intelligence for Interactive Digital Entertainment*.

LaValle, S. M., and Kuffner, J. J. 2001. Randomized Kinodynamic Planning. *The International Journal of Robotics Research* 20(5):378–400.

Smith, A. M., and Mateas, M. 2010. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. 273–280. Ieee.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. In *Proceedings of the Seventh International Conference on the Foundations of Digital Games*.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational Support for Play Testing Game Sketches. In *Proceedings of the Fifth International Conference on Artificial Intelligence for Interactive Digital Entertainment*.

Sorenson, N., and Pasquier, P. 2010. The Evolution of Fun : Automatic Level Design through Challenge Modeling. In *Proceedings of the International Conference on Computational Creativity*, 258–267.

The Open Motion Planning Library (OMPL). 2010. http://ompl.kavrakilab.org.

Togelius, J., and Schmidhuber, J. 2008. An experiment in automatic game design. In *IEEE Symposium On Computational Intelligence and Games*, 111–118.

van Dongen, S. 2000. *Graph Clustering by Flow Simulation*. Ph.D. Dissertation, University of Utrecht.