The Gold Standard: Automatically Generating Puzzle Game Levels

David Williams-King and **Jörg Denzinger** and **John Aycock*** and **Ben Stephenson**

Department of Computer Science, University of Calgary
2500 University Drive NW, Calgary, Alberta, Canada T2N 1N4
{dcwillia,denzinge,aycock,bdstephe}@ucalgary.ca

Abstract

KGoldrunner is a puzzle-oriented platform game with dynamic elements. This paper describes Goldspinner, an automatic level generation system for KGoldrunner. Goldspinner has two parts: a genetic algorithm that generates candidate levels, and simulations that use an AI agent to attempt to solve the level from the player's perspective. Our genetic algorithm determines how "good" a candidate level is by examining many different properties of the level, all based on its static aspects. Once the genetic algorithm identifies a good candidate, simulations are performed to evaluate the dynamic aspects of the level. Levels that are statically good may not be dynamically good (or even solvable), making simulation an essential aspect of our level generation system. By carefully optimizing our genetic algorithm and simulation agent we have created an efficient system capable of generating interesting levels in real time.

Introduction

Levels for puzzle games have traditionally been created by human level designers. More recently, researchers have begun automatically generating levels for puzzle games, both to reduce the burden on human designers while still keeping the game fun, and because level generation is an interesting and challenging problem. Automatic generation may also be important commercially, as it provides additional replay value by varying levels each time they are played.

We focus on KGoldrunner (Wadham and Krüger 2003), a non-scrolling platform puzzle game. Briefly, the player controls a hero who can walk around, fall from as high as necessary without injury, climb ladders and travel across monkey bars. The objective is to collect all the gold nuggets, at which point hidden ladders appear, and then escape by reaching the top of the screen.

The most important gameplay element of KGoldrunner is digging. When standing on a row of brick cells, the hero can dig away the cell below and to the left, or below and to the right. Digging holes allows the hero to drop through to previously inaccessible sections, and to trap and slow down the enemies that move toward the hero. In addition, holes eventually close up; if enemies are inside at that moment, they will die and respawn.

Many puzzle games are NP-hard (Viglietta 2012), including the 1983 classic Lode Runner (Smith 1983) on which KGoldrunner is based. Furthermore some puzzle games, like KGoldrunner, include dynamic elements which cannot be evaluated statically. Examples of such dynamic elements in KGoldrunner include enemies that move in response to the player and holes that fill in over time. The search trees created when generating levels are huge, and it is often impossible to explore them fully. Intelligent search heuristics are required to slice through the search space and find a level that is solvable, that is non-trivial, and hopefully, that is fun.

Although there are many papers that describe level generation for puzzle games with good heuristics, search methods, and data representations, few consider an obvious level evaluation: to simulate a user playing the level all the way through. This is, after all, what a level is designed for. Such a simulation can give very useful information about the solvability, difficulty, and time to solve a level.

In this work we use simple player simulations to create an efficient level-generation system for KGoldrunner, which we have named Goldspinner. Goldspinner uses a genetic algorithm to breed levels, where static analyses are used along with multiple player simulations to evaluate generated levels. The resulting system generates complex solvable levels, quickly enough that the next level can be generated as the player solves the current level. Goldspinner can also start with levels that have been partially human-created, making it useful as a design tool, too.

Related Work

A lot of work has been performed previously on procedural content generation (Togelius et al. 2011) and level generation for platform games, especially Super Mario Bros (Smith et al. 2009; 2011; Sorenson and Pasquier 2010a; Jennings-Teats, Smith, and Wardrip-Fruin 2010; Compton and Mateas 2006). A variety of different techniques have been used, with some systems incorporating the player's input (Nygren et al. 2011; Yannakakis and Togelius 2011), or a human level designer's input (Mawhorter and Mateas 2010). Our work fits most closely in the "simulation-based fitness function" category of Togelius et al. (2011), but we use our simulation for determining solvability, and in our case the

*Supported in part by a grant from NSERC.

level generation itself requires this simulation information.

Previous work has also explored techniques for creating levels for puzzle games like Sokoban (Murase, Matsubara, and Hiraga 1996). This work is particularly applicable to level generation for KGoldrunner because each game must consider cascading level modifications. Other related work has generated levels for several kinds of games (Sorenson and Pasquier 2010b), some with puzzle components. However, their approach is too slow for real-time level generation, making it impractical for our purposes.

Finally, several papers describe the genetic algorithm techniques that we use in our work (Hong, Huang, and Lin 2002; Kimbrough et al. 2002).

Level Generation Overview

KGoldrunner is a challenging game to generate levels for because of the dynamic nature of the gameplay. Bricks can be dug away, but they soon fill up again. Careful timing is necessary for the player to be successful at trapping enemies or escaping from them. Sometimes the player must rely on an enemy to carry gold into a reachable location. These factors combine to make static level analysis alone inadequate for KGoldrunner. Thus, while we do as many static analyses as possible, we augment them with player simulations that evaluate the dynamic aspects of the game.

Our level generation strategy consists of two parts. First, we use a genetic algorithm to breed levels that seem to be good based on static analyses. A variety of static analyses are performed which vary in complexity. Simpler analyses filter out undesirable levels so that more complex analyses are only performed on stronger candidates.

Second, we simulate gameplay, using an AI agent as the player. This determines both whether or not the level is solvable and estimates the difficulty of the level. While our static analysis can guarantee that the level will have many desirable properties, like any static analysis, it is unable to guarantee that the level is solvable because the game includes dynamic elements. During simulations, the AI agent interacts with the dynamic aspects of the game. Our AI agent collects gold nuggets, digs bricks to open new paths, and traps enemies in newly created holes.

The following sections describe each part in more detail.

Genetic Algorithm

The first part of the level-generation system is a genetic algorithm that breeds individual levels. Each individual consists of a base level (which may be empty or created by a human designer) and a list of "features" such as walls and nuggets that are overlaid on this level. The initial population consists of one individual with the base level and no additional features. Most of the genetic operators below take individuals and work on their lists of features, removing, adding, and splitting as necessary.

Plague. When our population reaches 50 individuals, a plague occurs. All levels below the 40^{th} percentile as measured by the evaluation function are discarded, reducing the population to 20 individuals. (All parameter values in Goldspinner were determined through experimentation.)

Bake. For each individual, the feature list for that individual is "baked" into its base level and then emptied. Baking the features into the level prevents them from being changed by subsequent crossover and mutation operations. Large feature sets negatively impact performance, making this operator necessary in order to generate levels in real time. A bake operation is performed every 500 generations.

Simplify. Occurs 12.5% of the time (assuming the special operators, plague and bake, are not applied). This operator removes a random number of features from an individual.

Crossover. Occurs 29.2% of the time. This operator selects one individual at random, and one individual at random from the upper half of the population as measured by the evaluation function. The feature list of each parent is split at a random point, and the left half of one parent's list is combined with the right half of the other parent's list. The base level of one parent is selected at random and the newly generated feature list is added to it to form a new individual.

Mutation. Occurs 58.3% of the time. An individual in the population is copied and mutated in one of four ways, adding a new individual to the population:

- If the level feature list includes at least one element, there is a 25% chance that a random feature will be removed.
- If no feature is removed, then there is a 50% chance that a new "wall" will be added to the level. The wall may be horizontal or vertical, with each direction being equally likely; wall placement and length are randomly chosen. The wall will be made of ladders (for vertical) or monkey bars (horizontal) 25% of the time; diggable brick, 37.5% of the time; or concrete, 37.5% of the time.
- If no feature is removed and no wall is added, there is a $\frac{1}{6}$ chance that an enemy will be added at a random location.
- Otherwise, a nugget will be added at a random position.

The process of applying operators repeats, picking precisely one operator at each generation, until the specified number of generations has passed. The highest-evaluated level at that point is the final level. The evaluation function is described in the next section.

Evaluation Function

In order to calculate the evaluation function we build a graph, which we call the *group graph*, describing how the player can move within the level. In addition, a second version of the group graph, the *hidden group graph*, allows us to determine the same properties once any hidden ladders are revealed after the last nugget is collected. When it is possible for the hero to travel between two nodes, they are connected by a directed edge in the graph.

Each node in the group graph represents a non-empty set of cells in the level that are related to each other in one of three ways: commutative, freefall, and diggable.

Commutative. The hero can move backward and forward between any two adjacent cells in the node.

Freefall. Once the hero enters the group, the hero will fall downward. The hero can only exit the group from the cell closest to the bottom of the screen.

Diggable. All of the bricks in this group can theoretically be removed by digging. When Diggable groups are identified they assume that the hero is capable of reaching all necessary open areas around the node, and that the holes that are dug never disappear. In practice it is generally not possible to dig every brick identified in the Diggable group. The simulations we perform ultimately determine whether or not all of the bricks necessary to complete the level can be removed.

Our evaluation function begins by examining the static characteristics of the level. Later, simulations are performed if the static characteristics are sufficiently strong to justify doing so. A level's evaluation value is calculated as follows:

1. The evaluation value, E, is initialized to 0.0.
2. A flood fill is performed from the starting location to identify all areas that can be reached without passing through concrete (non-diggable walls). The flood fill considers all directions, even straight upwards. If there are nuggets that are not reached by the flood fill, or the top of the screen is not reachable, then the level is definitely not solvable, so a negative evaluation is returned immediately.
3. A breadth-first search of valid moves from the initial player position is considered, where brick walls are considered passable. If all the nuggets can be reached in this manner, followed by the top of the screen, then 1000 is added to E. This search does not consider enemy movement, hole creation, or holes' refill rate.
4. Using the group graph, we determine whether every nugget position is *forward-reachable* in the group graph, and *backward-reachable* in the hidden group graph. If so, 10,000,000 is added to E because this is a very good indication the level is solvable. Forward-reachable means the hero can get to the position, and backward-reachable means that from the position, the exit locations are not all cut off. These are necessary conditions unless enemies can carry gold to the hero.
5. If the cell corresponding to the start position is backward-reachable in the hidden group graph, then 1,000,000 is added to E, because the top of the screen is reachable once hidden ladders are exposed.
6. If the number of features is less than or equal to 10 then 10 times the number of features in the level is added to E. If the number of features exceeds 10 then 110 minus the number of features in the level is added to E, giving diminishing values as the number of features grows.
7. If the number of enemies in the level is nonzero, then 250 minus fifty times the number of enemies is added to E.
8. Finally, 30 times the number of nuggets in the level is added to E, to heavily favor levels with more nuggets. Nuggets are also counted as features in step 6.

This completes the static part of the evaluation function. If E is at least 11,000,000 — meaning that all the group-graph tests indicate that the level looks solvable — then simulations are run on the level as described below. The simulations' results are incorporated into E as follows:

9. If the number of successful simulation plans (where twenty in all are run) is less than three, then $1e9$ times the number of successful plans is added to E. Otherwise, if the number of successes is at least three, then $3.3e9$ minus $1e8$ times the number of successes is added to E.

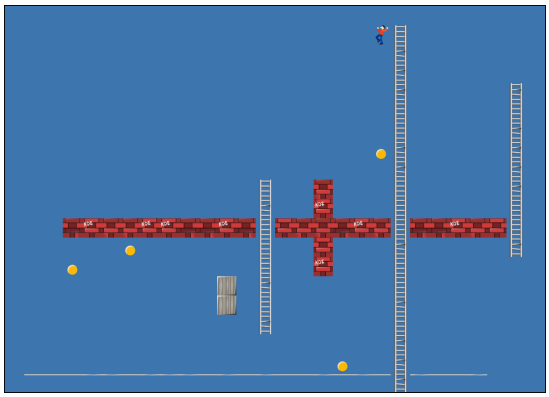

Figure 1: A level constructed using only static analyses.

Using static analyses alone can result in interesting looking levels. For example, the level shown in Figure 1 was created after 1000 generations of static analyses. It includes a mixture of ladders and monkey bars, many diggable bricks, and four nuggets that must be collected by the player. The level scores very well in the evaluation function ($E = 1.10011 \times 10^7$), because the group-graph indicates the hero might be able to pass through the upright brick wall in the centre; however, dynamic simulation shows that the level is not in fact solvable.

Simulations

Levels that are generated by the genetic algorithm, and which pass static tests, are subjected to an additional evaluation step. To see whether a level is solvable, Goldspinner generates a series of high-level global "plans" based around the group graph, and runs a full simulation of the game for each plan to see whether any of the plans are successful. If no plans work, we consider the level unsolvable; if too many plans work, the level may be too easy. Plans are constructed as follows.

The Nugget Graph and Clique Graph

There are three graph structures, shown in Figure 2, involved in the construction of a plan. Starting from the group graph, the first step is to construct a *nugget graph*. The nugget graph nodes are the subset of nodes in the group graph which contain one or more nuggets; there is an edge between nugget nodes if there is a path of any length between those nodes in the original group graph.

Nugget graph → clique graph. From the nugget graph, we construct the *clique graph*, which is based on the nugget graph but has any cliques[1] compressed into one node. We

[1]A clique is a completely connected subgraph. As the nugget graph is transitively closed, detecting cliques is at most $O(n^2)$.

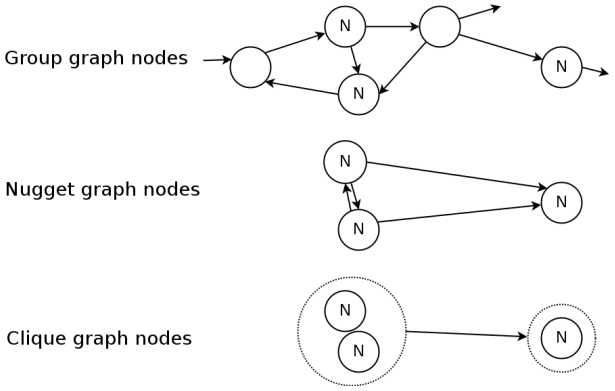

Figure 2: Graph construction ("N" is a nugget node).

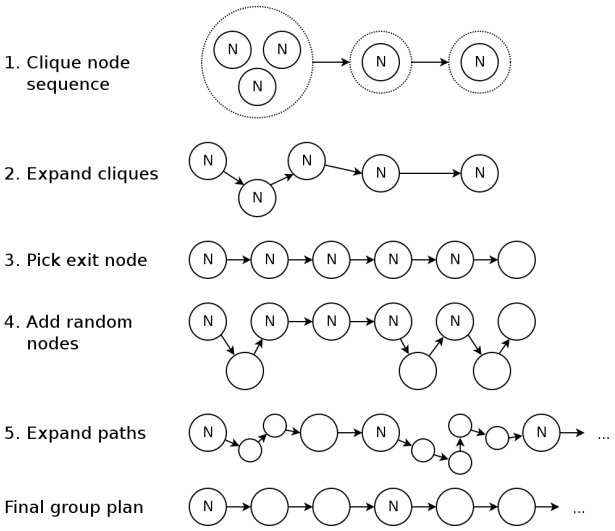

Figure 3: The steps to building the group plan.

examine the clique graph to ensure it is linear, i.e., its nodes can be laid out in a connected line (with additional edges to all nodes further forwards, since the graph is transitively closed). This is done by finding a path of length n (where n is the number of nodes in the clique graph). If the graph is linear, the player will be able to pick up all the nuggets.

Clique graph → group plan. Starting from the path in the clique graph, which can be reused as necessary for any one level, we generate 20 plans as described below and simulate each one in full. The initial path goes through several steps to reach a *group plan*. This is later expanded to form a *concrete plan*, which is relatively easy for our AI agent to follow in the simulation. The group plan is constructed as follows (see Figure 3):

1. First, start with a sequence of clique graph nodes: the path of length n mentioned earlier.
2. Next, turn the clique graph into a sequence of nugget nodes by randomly selecting a permutation of the nodes within each clique and replacing the clique node with that permutation. This is the order in which the plan will attempt to visit nugget nodes.
3. An extra node must be added at the end of the sequence which is an exit node. The normal group graph must be used until the last nugget node has been reached, at which point the hidden group graph must be used instead. There will exist some valid exit node because each nugget node is backwards-reachable in the hidden group graph.
4. Add some randomness to the sequence by performing the following between zero and two times: pick a random node, r, and its successor, s, in the sequence. Select a node n such that r can reach n and n can reach s. Insert n between r and s so that the sequence now reads r, n, s. If no such node n exists, skip this round.
5. Now the sequence consists of elements with arbitrary paths between them, but we need a sequence of group graph nodes that are directly connected. So for each r and s in order in the sequence, find a random group-graph path of minimal length,[2] and expand the sequence with those nodes.

This gives the group plan, which outlines the order in which group graph nodes will be visited. There are a few additional steps that must take place to form the concrete plan for the AI agent to use, however.

Group plan → concrete plan. For each r and s in the sequence, where there is an edge $r \rightarrow s$, select a random pair of exit/enter cells such that the exit cell is in r, the enter cell is in s, and the hero can move directly from the exit cell to the enter cell. (The set of these valid exit/enter cells is computed per edge in the group graph upon the group graph's construction.) The plan will require the hero to exit and enter each node at these positions. The first node does not need an enter cell, as the hero's starting position is known, but the last node does need an exit cell, which can be anything along the top row of the level.

Finally, each (concrete) node is given a set of goals that must be visited by the hero before going on to the next node. All the nuggets in the level are added as nugget-goals to the enclosing nugget node, the first time it occurs in the sequence of nodes (since paths between other nodes might well revisit a nugget node again). After this, dig-goals are added to nodes which transition into diggable group graph nodes; the dig-goals are such that if the hero satisfies all of them, the transition to the next node will be possible.[3]

This gives the concrete plan, which is used directly in the simulation of the game, described next.

Game Simulation

Once a concrete plan has been established, game simulation takes place with an AI agent that tries to follow the plan.

We carefully examined the source code for KGoldrunner while we were creating our simulation. We used exact millisecond timings from the real game for movement speeds and dug-brick times. We copied the enemy AI directly into our program, using an interface that maps our data structures

[2]This can be done with a randomized breadth-first-search, which adds successors to any node visited in random order.

[3]The system can only currently dig through one layer of brick.

to the ones present in the KGoldrunner source, so that each simulated enemy behaves exactly as it would in the game. Overall, we try to have an accurate frame-by-frame simulation of the game.[4]

In this game simulation, the AI agent is consulted each time the hero must make a new move. The agent follows the plan as closely as it can: if it moves outside the current group graph node, the plan is considered a failure. Each node is solved by finding a current path as follows:

1. For each nugget goal, in order, find a path to the goal.
2. Similarly, for each dig goal, find a path to a cell that can dig away the goal (and then dig it).
3. Otherwise, find a path to the current node's exit cell.

The first such path that is found is set as the current path, and the agent will follow precisely those moves until the goal is reached; then another goal is sought. If a goal cannot be reached, the plan is considered a failure. As soon as an exit cell is reached, the next move is automatically the transfer to the enter cell of the next node.

Given enough plans, the system described above will often find a way to avoid enemies through a clever ordering of group-graph nodes and enter/exit cells. However, if the AI agent encounters an enemy on its chosen path, it will attempt to trap the enemy by digging a hole and waiting for the enemy to fall in so the agent can run over top of the enemy and continue along the path. As a side effect, any gold the enemy may have picked up will be collected by this action.

Results

We ran Goldspinner numerous times with different parameters to measure its efficiency. This section also shows, anecdotally, that Goldspinner generates interesting levels. A full user evaluation, to see how many levels are actually "fun", is part of future work.

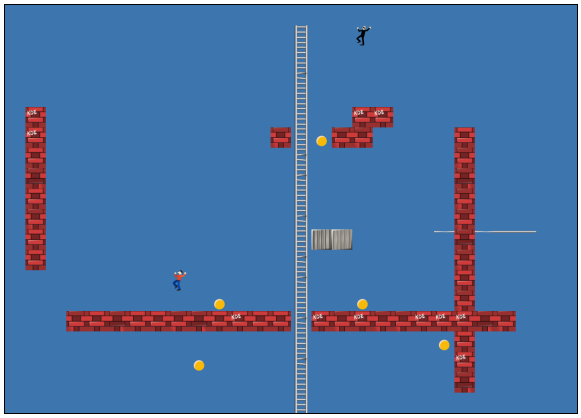

Figure 4: Here, the enemy must be trapped at least once.

Figure 4 shows a level where the enemy gets in the way of the player and must be trapped at least once. This level also demonstrates the uncertainty that makes static analysis difficult: will the enemy go left and get the gold, or not? Where will the player need to trap the enemy?

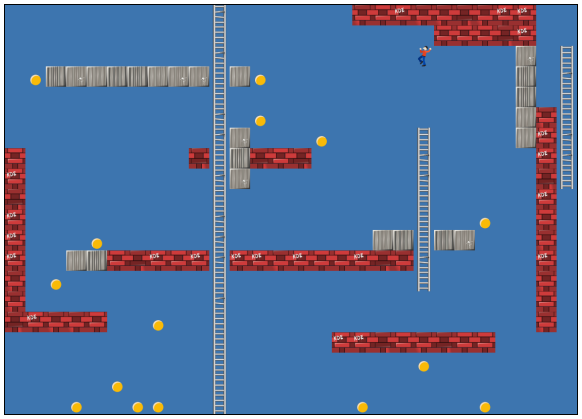

Figure 5: Typical level, with lots of gold nuggets to pick up.

In Figure 5, a level generated from scratch in 5000 generations, there are many nuggets that must be picked up. The gold nugget below and to the right of the hero must be retrieved first, because the hero cannot get back to that spot. The clique graph makes this immediately apparent to the simulations, which always go for that nugget first.

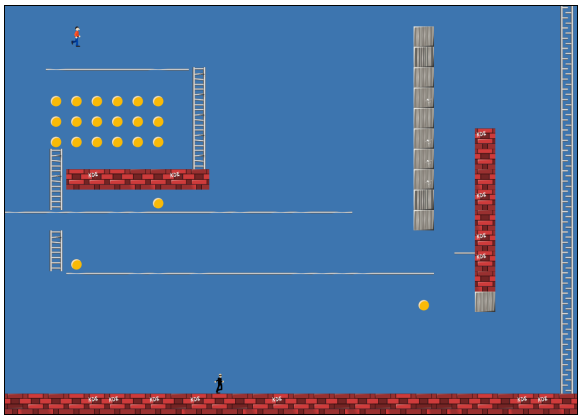

Figure 6: Level generated from a "block of gold" base level.

Finally, the large block of gold in Figure 6 and its surrounding bar, ladder, and brick were created by a human designer (along with a few other features, including a hidden ladder on the right-hand side of the level). Goldspinner took the base level and filled in more features, including an enemy at the bottom, ending up with a solvable level that very closely follows the structure of the original base level.

Performance. In order for an automatic level generation system to be useful it must generate interesting levels in a timely manner. This section shows that our system meets this need. All of our performance tests were conducted using one core of a 3.16GHz Intel Core 2 Duo processor running Scientific Linux 6.1. Our system was developed in C++, and was compiled with optimizations enabled (g++ -O2).

[4]We mostly followed KGoldrunner's Traditional rule set. There are features we did not implement, however, such as a rare case preventing two enemies from ending up in the same cell.

The following shows timing information with Goldspinner starting from an empty base level:

Generations (1000 runs each)	Percent solvable *static*	*dynamic*	Generations per second
1000	76.50%	10.90%	822.91
2000	90.20%	31.70%	257.33
3000	92.90%	36.90%	141.31
4000	96.10%	44.20%	97.32
5000	97.10%	44.70%	71.25

"Percent solvable" refers to how many runs yielded a level that was statically and dynamically solvable, respectively.

Running Goldspinner for more generations takes longer but will generate more complex levels, and levels that are more likely to be solvable. Goldspinner can be run multiple times with a low number of generations and find a dynamically solvable level in a matter of seconds; or a smaller number of instances can be run with more generations (like 5000), and a good level can be found in a few minutes. The standard deviation on the runtimes is high, but Goldspinner can be terminated at any point and it will print the best level it has found so far. This is easily fast enough, especially if parallelized, to generate a level while the player is playing a previous level, meeting our definition of "real-time".

The following shows timing information with Goldspinner starting from different base levels, at 2000 generations in each case:

Base level (400 runs each)	Percent solvable *static*	*dynamic*	Generations per second
1 ladder on right	99.75%	92.75%	63.06
1 hidden ladder	96.50%	92.25%	63.77
two ways	100.0%	73.50%	42.27
block of gold	100.0%	94.50%	33.36

If the initial base level is not dynamically solvable, Goldspinner takes a while to find a solvable variation and then refine it from there. This is why "two ways", the only base which was not solvable, has a lower dynamic solvability percentage. So a human level designer using Goldspinner would get better results by supplying the program with a solvable base level (as in Figure 6); if this is not done, however, Goldspinner can still find ways to generate good levels.

Conclusion and Future Work

We have described a two-part process for generating levels for puzzle games that include a dynamic element. Our technique employs a genetic algorithm that initially evaluates static properties of the level. When the static properties of the level are sufficiently strong, additional resources are invested and full simulations of the level are performed using an AI player. This ensures that the generated level is solvable (a property that cannot generally be guaranteed from static analysis alone when dynamic game elements are present) while also allowing other desirable static and dynamic properties to be considered.

We have applied this technique to KGoldrunner, a puzzle based game with dynamic elements such as enemies and holes that refill over time. We have shown that our system generates interesting, appropriately complex levels, and that these levels are guaranteed to be solvable. Our performance results have also shown that these levels are generated in a reasonable amount of time, allowing a new level to be generated as the player works to complete the current level.

While Goldspinner currently generates interesting levels of appropriate complexity, we believe it can be further improved. Our current implementation gathers additional data during simulations such as the time for completion and the number of cells traversed, data that we do not use at present. Feeding that into the evaluation function along with further tuning could yield even more interesting levels.

Finally, we conjecture that our technique may be generalized to other games with dynamic elements.

References

Compton, K., and Mateas, M. 2006. Procedural level design for platform games. In *2nd AIIDE*, 109–111.

Hong, T.; Huang, K.; and Lin, W. 2002. Applying genetic algorithms to game search trees. *Soft Computing* 6(3):277–283.

Jennings-Teats, M.; Smith, G.; and Wardrip-Fruin, N. 2010. Polymorph: A model for dynamic level generation. In *6th AIIDE*, 138–143.

Kimbrough, S.; Lu, M.; Wood, D.; and Wu, D. 2002. Exploring a two-market genetic algorithm. In *Genetic and Evolutionary Computation Conference*, 415–422.

Mawhorter, P., and Mateas, M. 2010. Procedural level generation using occupancy-regulated extension. In *IEEE CIG*, 351–358.

Murase, Y.; Matsubara, H.; and Hiraga, Y. 1996. Automatic making of Sokoban problems. *PRICAI'96: Topics in Artificial Intelligence* 592–600.

Nygren, N.; Denzinger, J.; Stephenson, B.; and Aycock, J. 2011. User-preference based automated level generation for platform games. In *IEEE CIG*, 55–62.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2D platformers. In *4th Int. Conf. on Foundations of Digital Games*, 175–182.

Smith, G.; Whitehead, J.; Mateas, M.; Treanor, M.; March, J.; and Cha, M. 2011. Launchpad: A rhythm-based level generator for 2-D platformers. *IEEE Transactions on Computational Intelligence and AI in Games* 3(1):1–16.

Smith, D. E. 1983. Lode Runner. Brøderbund.

Sorenson, N., and Pasquier, P. 2010a. The evolution of fun: Automatic level design through challenge modeling. In *1st Int. Conf. on Computational Creativity*, 258–267.

Sorenson, N., and Pasquier, P. 2010b. Towards a generic framework for automated video game level creation. *Applications of Evolutionary Computation* 131–140.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.

Viglietta, G. 2012. Gaming is a hard job, but someone has to do it! In *6th International Conference on Fun with Algorithms*. Forthcoming.

Wadham, I., and Krüger, M. 2003. KGoldrunner. <http://www.kde.org/applications/games/kgoldrunner/> Retrieved 29 Sept 2011.

Yannakakis, G., and Togelius, J. 2011. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing* 2(3):147–161.