Evolving Playable Content for Cut the Rope through a Simulation-Based Approach

Mohammad Shaker[1], **Noor Shaker**[2] **and Julian Togelius**[2]
[1]Faculty of Information Technology Engineering, Damascus, Syria
[2]Center of Computer Game Research, IT University of Copenhagen, Copenhagen, Denmark
{mohammadshakergtr}@gmail.com, {nosh, juto}@itu.dk

Abstract

In order to automatically generate high-quality game levels, one needs to be able to automatically verify that the levels are playable. The simulation-based approach to playability testing uses an artificial agent to play through the level, but building such an agent is not always an easy task and such an agent is not always readily available. We discuss this problem in the context of the physics-based puzzle game *Cut the Rope*, which features continuous time and state space, making several approaches such as exhaustive search and reactive agents inefficient. We show that a deliberative Prolog-based agent can be used to suggest all sensible moves at each state, which allows us to restrict the search space so that depth-first search for solutions become viable. This agent is successfully used to test playability in *Ropossum*, a level generator based on grammatical evolution. The method proposed in this paper is likely to be useful for a large variety of games with similar characteristics.

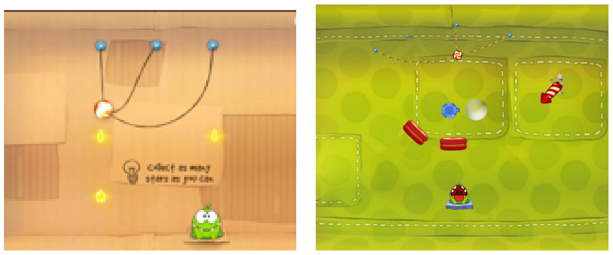
1 Introduction

Procedural generation of game content, such as maps, levels, items and quests, is currently one of the most active fields of research within CI/AI in games research, motivated by a real need within the games industry as well as by scientific curiosity regarding what types of content can be generated and what we can do with this technology (Togelius et al. 2010b; Yannakakis 2012). One of the key challenges when generating game content is to evaluate the quality of the generated artefacts. One particular kind of quality assurance that is essential for generation of "necessary" game content such as levels is playability control, i.e. ascertaining that the level can be successfully played through by a human player. Unplayable levels could be such things as StarCraft maps where there are no paths between bases, or Super Mario Bros levels which contain gaps that are too wide to jump over. In many games, a boring level is undesirable but acceptable, whereas a level that cannot be passed breaks the game completely.

While it is possible to build constructive (single-pass) level generators for some games that always produce playable levels without playability checking, this is typically done at great expense of the expressive range of the generator. Search-based procedural level generators, that use some kind of optimisation algorithm to search the space of possible game levels, rely either on the representation to preclude unplayable levels (again, reducing the expressive range of the generator), or on the evaluation function to check the playability of the level. Therefore, several recent search-based level generators have included playability checks in their evaluation functions.

Playability checking could be done in different ways. There are more or less direct approaches, based on e.g. verifying the existence of paths between different points (Togelius et al. 2010a), measuring gap widths and drop heights (Smith, Whitehead, and Mateas 2010) or proving the existence of solutions when the game mechanics or some approximation thereof can be encoded as first-order logic (Smith and Mateas 2011). However, in many cases you need to actually play the level in order to show that it is playable, and thus requires us to build an agent capable of playing the game (Ortega et al. 2012; Jaffe et al. 2012). (Note that for any interesting games this can only ever be a negative proof; if an agent is not able to play a level, this does not necessarily mean that it is unplayable.) Building an agent capable of proficiently playing a given game is rarely straightforward.

The most straightforward approach to building a game-playing agent is to handcode it – however, this approach is very labour intensive, non-portable, and introduces biases towards play-proofing levels that fits with that agent's playing style. Another approach is to use some form of reinforcement learning algorithm to learn to play the game, but this approach suffers from that it's typically hard for an algorithm to quickly learn to play a game well (Togelius and Schmidhuber 2008). One can use some form of tree search algorithm to play the game – this works well for testing content such as board game rules (Browne and Maire 2010), but is more problematic when testing content such as levels for continuous-time games, with large branching factors and huge state spaces such as the one we are dealing with. Exhaustive search and most varieties of tree search are unfeasible in such games due to the huge state space (the number of states that would need to be evaluated for an average length level in our testbed game is $430 * 10^{98}$). Reactive planing approaches can also be applied but they suffer from their limited planning ability and training them is not obvious due to the nature of the game which requires informed



(a) (b)

Figure 1: Two snapshots from the original Cut The Rope game (c) and our clone version of it (d) showing Om Nom waiting for the candy which is attached to ropes.

choice of actions and precise timing when performing them.

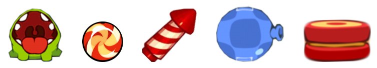
In this paper, we consider the problem of automatically generating playable levels for the popular physics puzzle game *Cut the Rope*[1]. In a previous paper, we described *Ropossum*, a program that generates levels for this game using grammatical evolution and a heuristic evaluation function (Shaker et al. 2013). While that approach generated interesting levels, it did not strictly guarantee that they were solvable, and concerns of solvability in the heuristic function limited the range of levels that could be generated. In this paper, we describe a method for automatically playing Cut the Rope levels and the integration of that method into our level generator. The method is based on depth-first search in state space using simulations of the game by the original physics engine. As it would be computationally intractable to simulate the game at a fine granularity using this method, we have restricted the search to only investigate *sensible* search paths by: (1) encoding core components of the game rules into a Prolog-based agent; (2) incorporating as much information about the context by introducing the set of *reachable components* and (3) defining the branching points of the state tree by only the possible actions returned by this agent.

2 Cut The Rope

The testbed game chosen is a clone of *Cut The Rope* (CTR), a popular commercial physics-based puzzle video game released in 2010 by ZeptoLab for iOS and Android devices. The game was a huge success when released and, at the time of writing, it has been downloaded more than 100 million times. There is no open source code available for the game so we had to implement our own clone using a heavily modified version of the CRUST engine (Millington 2007) and the original game art assets. The modifications include adapting the engine to work in a 2D environment and implementing the spring constraint. Our clone of the game, called *Cut The Rope: Play Forever*, features most of the fundamental characteristics of the original games. Figure 1.(c) shows one of the level in the original game while Figure 1.(d) presents a level from Cut The Rope: Play Forever.

The gameplay in CTR revolves around feeding candy to

[1]Copyright 2010 ZeptoLab



(a) Om Nom (b) Candy (c) Rocket (d) Air-cushion (e) Bumper

Figure 2: The various components presented in the original *Cut the Rope* game and considered in our clone.

a little green monster named *Om Nom*. The candy is usually attached to one or more ropes which has to be cut with a swipe of the finger in order to set it free. All game objects obey Newtonian physics and are affected by gravity. The player looses the game by letting the candy escape (e.g. fall) outside the level boundaries. The game features a puzzle component by the presence of obstacles and other physics-based components that help redirecting the candy. The set of components included in the original game includes air-cushions, constrained pins, bubbles, shooting-buttons, rockets, spikes, spiders among others (see Figure 2). The player interacts with the game by cutting a rope, tapping an air-cushion, a bubble or a button triggering a sequence of physics-based consequences. Solving the level puzzle depends to a great extent on timing. Specific actions should be taken in certain game states, otherwise the player looses the game.

3 Game Design and Grammar

When developing a Design Grammar (DG) for Cut the Rope levels, we chose to focus on a design pattern of the levels that consists of ropes, air-cushions, bumpers, bubbles and rockets. These are the most of the basic components that could be presented in a level and therefore this pattern allows generating interesting combinations and permits meaningful exploration of the content space. Detailed information about these (and some other) components and their properties can be found in (Shaker et al. 2013).

Evolving the design of the levels is done using grammatical evolution.The structure of the levels is represented in a design grammar used by GE to evolve the levels. For more information about how GE works in the context of level generation and more through analysis of the DG specified for CTR, the reader may refer to (Shaker et al. 2013).

Design Contraints

According to the design grammar defined, the different components can be placed at any position in the level map. Although this allows for exploring a wide space of possible combinations, not all of these configurations are interesting in terms of playability and aesthetics considerations. Therefore, we define a set of conditions that should be satisfied in the final design. A penalty is associated with each violation of these conditions according to their importance and a score is assigned for the level design as a linear combination of the conditions according to the equation:

$$score = 25 * P_{candy} + 10 * P_{OmNom} + 10 * N_{air-cushion} * P_{air-cushion} + 20 * N_{rocket} * O_{rocket} + 10 * N_{bubble} * P_{bubble} + 25 * C_{overlap}$$

where N_x represents the number of objects of type x presented in the level, P_{candy}, P_{OmNom}, $P_{air-cushion}$, P_{bubble} define a set of constraints on the placement of the candy, Om Nom, air-cushions, and bubbles, respectively, O_{rocket} controls the direction of rockets, C_{plac} defines a constraint of a predefined distance preserved between the components and $C_{overlap}$ is the number of overlapped components. More information about each of these conditions can be found in (Shaker et al. 2013). The final score is assigned as part of the fitness for each level design evolved while the other part concerns the result of cheating for playability as discussed in the following sections.

4 Constrained Search with Rule-based Agent

The main contribution in this paper is simulation-based playability testing for a physics-based game using an artificial agent. In this section we present the design of this agent. Since CTR is a physics-base game, all the components adhere to basic physics such as gravity, collision, bumping, air blowing and floating. To solve the game, the player should make informed decisions about what action to perform and when. The possible actions that can be performed each time step are the following:

1. Rope cut: this action can be performed on any rope in the level. Only one rope can be cut at a given timestep.
2. Air-cushion press: the air-cushion can be pressed at any time, however, the candy will be affected only when it is close enough to the opening of the air-cushion.
3. Bubble burst: a bubble can be burst only when containing the candy.
4. Rocket press: when the candy becomes within a close distance to a rocket, they become attached and the rocket starts moving. To detach them, a press action should be performed on the rocket. This results in setting the candy free in a direction depending on the rocket's direction and speed and the gravity force.
5. Void: the player can chose not to perform any action for a number of timesteps waiting for the candy to reach a specific position.

Several different approaches to developing a proficient game-playing agent for Cut the Rope were attempted during this work, including evolving action sequences and reactive agents, but with unsatisfactory results due to their limitations presented previously.

The approach we settled for, and which we are presenting in this paper, is to use a Prolog-based agent doing simple inference to identify *sensible actions*, and do a depth-first search for a strategy that can solve the level using only those actions generated by the agent. The search process unfolds as follows: the physics engine starts the simulation. At each time step, a description of the game state represented as facts (see figure 3 for an example) is passed to the Prolog-based agent. The agent returns all sensible actions at that state – in many cases, the only sensible action is "void", but in other states two or more actions are sensible. If more than one action is returned, the search branches and all actions are explored. This process diminishes the search space considerably and allows us to search through all sensible strategies to solve the level in an acceptably short time. However, it requires that the agent identifies a good set of actions at each step, a challenge which we solve by encoding key game mechanics and micro-strategy as Prolog clauses (rules). Below, we describe two different rule sets and highlight the limitations and advantages of each.

```
candy(527, 140). velocity_up. velocity_to_right.
frog(390, 330). rope(550, 50, 105).
rope(440, 50, 195). air_cush(507, 195, 4).
reachable(rocket).
```

Figure 3: Example of the facts representing a game state.

Ruleset 1: Properties and Placement

In our first attempt to specify the agent's ruleset we focused of the components' properties and their relative placement in the level map as indicators of the next action to perform. The ruleset is included as figure 4. According to the figure, cutting a rope can be performed when the candy, attached to the rope, is within a predefined distance from Om Nom or when it is inside a bubble that is below Om Nom. The second clause specifies when to press an air-cushion. This can be done whenever the candy and the air-cushion are within a small distance and the candy is in the airflow direction. The bubble burst action can be performed according to the third rule which states that the candy should be inside a bubble, the bubble is in a higher position than Om Nom, both are located within a small distance and the candy is moving towards Om Nom. The fourth clause handles the action of firing a rocket. This is done when the candy is attached to the rocket and the rocket is above and close enough to Om Nom. The void action is always added to the set of sensible actions. The remaining clauses do necessary distance and direction calculations.

When an action is applied it is applied to all components of the specified type which it can be applied to; remember that bubbles must contain the candy to be poppable, ropes connected to the candy to be cuttable etc.

Action Priority Since it is possible that more than one action can be applied at a given timestep, a priority value is assigned for each of them. The values are defined based on the importance of the action and the component and are based on extensive play testing. The highest priority is given to (1) cutting a rope followed by (2) tapping an air-cushion, (3) firing a rocket, (4) bursting a bubble and (5) the void action.

The defined ruleset is intuitive, relatively simple to interpret and easy to implement. It has, however, a number of clear limitations. The decision about performing an action depends to a great extend on the thresholds specifying when two components are close enough. Although the values of the thresholds were assigned experimentally, the results showed that many potentially playable levels are misclassified because of violating these thresholds with small margins. Moreover, the rules defined consider only the positions, distances and directions of the components. It does

```
[1]rope_cut :- rope(_,_), Om_Nom(Xo,Yo)(
  (candy(Xc,Yc), distance(Xo,Yo,Xc,Yc,50))
  | (active_bubble(_,Yb),(Yf<Yb)))
[2]air-cush_press :- air_cush(X,Y,Dir),
  candy(Xc, Yc), distance(X, Y, Xc, Yc, 80),
  candy_in_direction_of_air_cush(X,Y)
[3]bubble_burst :- active_bubble(X,Y),
  OmNom(Xo, Yo), (Yo > Y), distance(X, Y, 50),
  ((velocity_to_right, OmNom_to_right(X, Y))
  |(velocity_to_left, OmNom_to_left(X, Y)))
[4]rocket_press :- active_rocket(X,Y,_),Yo < Y,
  OmNom(Xo, Yo), distance(Xo,Yo,X,Y,100)
[5]void_action
[6]candy_in_direction_of_air_cush(X,Y) :-
  candy(Xc,Yc), air_cushion(X,Y,Dir),
  ((Xc < X , dir_to_left(Dir))
  |(Xc > X, dir_to_right(Dir)))
```

Figure 4: The first rulesets defined to build an AI agent based on component placement and properties.

not take into account, for example, the overall structure of the level and the possible consequences of performing an action on winning or losing the game. An action, according to this ruleset, is performed when the game state allows taking this action accounting only for the minimum context information required. Testing also showed long processing time (an average of 79.6 seconds per level for 20 levels tested).

After testing the first ruleset on several designs, it became clear that, although being simple and clear and showing ability to solve some cases, the construction of more context-dependent approach would be more appropriate and could potentially solve the problems arise in our first design.

Ruleset 2: Physics-based Reasoning Agent

The main idea behind the design of the second ruleset is to consider as much as possible of the context information yet keep the processing time to a minimum. To allow this, we define the set of *Reachable Components*, RC, which contains the components the candy can reach while in a given position, direction and velocity. The next action to perform is calculated based on this set; if an action to perform leads to an empty set, the action is simply discarded because this means that the candy will fall outside the boundary of the canvas and the game will be over. The level is obviously playable when, at any timestep, the RC contains Om Nom. The result will be a smaller set of actions guaranteed to initiate a sequence of interactions with the other components. By using the RC set, the process of evaluating whether a level is playable becomes much faster since the search space is smaller containing only promising actions. Figure 5 presents different states while playing a level with the set of RC highlighted.

Figure 6 presents the second ruleset defined to deal with the physics properties of the components and handle the RC set. As in the first ruleset, the first five clauses define when to perform each action, but in this case, the action is taken based on the RC set. For example, a rope can be cut if freeing the candy will result in interactions with other components. The same principle applies to the bubble burst action. The conditions for pressing an air-cushion are similar to the ones in the first ruleset since it is hard to calculate the possible trajectory of the candy because this depends on several factors including its initial direction and speed and whether it is in a bubble and/or attached to a rope. A rocket is fired if it is carrying the candy and there is an intersection between the rocket trajectory and another component. Finally, the void action is always added to the set of possible actions as long as there exist at least one reachable component.

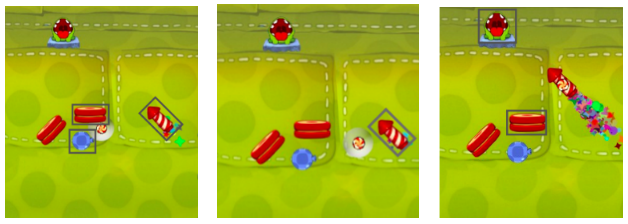

(a) (b) (c)

Figure 5: A level while being played. The set of reachable components is highlighted.

The detection of whether the candy can reach a certain position in the map is handled by the physics engine given the candy's current position, velocity and direction. If a component is placed within the range that the candy can reach, a fact is added in the game state indicating that the component is reachable in the form, $reachable(comp)$.

The same priority values of the actions described in the previous section are also considered in this ruleset.

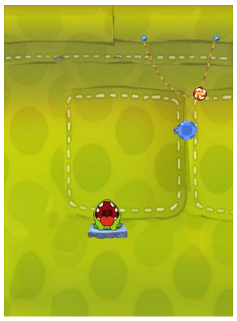
The initial results obtained by applying this ruleset showed very promising results and the comparison between the average processing time required showed that a level can be solved in only 29 sec compared to 79 sec required to solve a level following ruleset 1. More discussion and analysis of the results obtained are presented in Section 6.

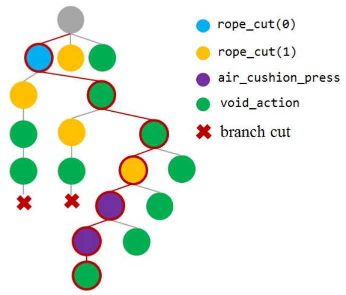
```
[1]rope_cut :- rope(_,_), reachable(_)
[2]air-cush_press :- air-cush(X,Y,Dir),
  candy_in_direction_of_air-cush(X,Y),
  distance(X, Y, 80)
[3]bubble_pinch :- active_bubble(_,_),
reachable(_)
[4]rocket_press :- active_rocket(X,Y,Dir),
reachable(_)
[5]void :- rope_cut | bubble_pinch
     | air-cush_press | rocket_press |
     reachable(_)
```

Figure 6: The second ruleset defined to describe the behaviour of the AI agent. The main advantages of this ruleset is the inclusion of the concept of reachable components.

5 Evolving Playable Content

Now that we have a method for representing level design and a procedure to assess whether a given design is playable,


(a)



(b)

Figure 7: Example level design and its corresponding game state tree. The actions taken are presented in different color. The sequence of actions that solve the game is highlighted in red.

we present a framework where these two methods are combined to evolve playable content. The next sections present a detailed description of the framework implemented. For the rest of the paper, we only present experiments and discussion based on the second ruleset since it demonstrated efficient exploring and handling for various cases.

Checking for Playability

The playability evaluation module consists of two parts: the physics engine (PE) and the inference engine (IE). These two communicate at each timestep through the following process: (1) the PE initiates the process by building a state tree of one root node corresponding to the current game state and (2) sending the game state to the IE as facts (see Figure 3 for an example), (3) given the information about game state, the IE infers the next possible action(s) to perform and send the set to the PE, (4) the PE orders the actions according to their priorities and for each action creates a new node in the tree corresponding to the state of the game after performing that action, (5) the PE starts traversing the tree in a depth-first approach exploring the subtree of each action by repeating the process starting form step 2. The depth-first approach is preferable because we are only interested in checking whether the level is playable rather than investigating all possible ways in which the level can be solved. The process is repeated until the game is found playable/unplayable or a maximum tree depth is reached.

Figure 7 presents a simple level structure and its corresponding state tree. Each node in the tree corresponds to a game state and the action taken at that state is presented. Different actions are presented in different colors and the sequence of actions that solve the level is highlighted. Note that some of the branches (marked with an x) are not fully explored because they lead to an empty RC set and therefore losing the game.

The Overall Framework

Grammatical Evolution is used to evolve level designed according to the design grammar defined. A score value is then assigned to each individual based on the design constrained presented in Section 3. The levels of high scores (higher than 75%) are then evaluated for playability through a simulation-based approach. For each acceptable level design evolved, the PE is initiated and several calls are made to the IE to detect whether the design is playable. If the level is playable, the evolution process is terminated. Otherwise, the fitness is calculated according to the equation:

$$fitness = 0.25 * score + 0.75 * dis$$

where $score$ is calculated as presented in Section 3 and dis is the smallest distance between the final position of the candy and Om Nom. These two values are normalized to the range [0,1] using min-max normalization.

Performance Improvement

The use of the RC in the second ruleset has a significant effect on improving the processing speed by completely discarding the branches in the tree that eventually lead to losing the game. We also investigated another method for improving the performance by applying an adaptive time step.

Adaptive Time Step The game state in the physics engine is updated 60 times per second which corresponds to 60 calls to the inference engine. However, most of these states should not be evaluated if we are to imitate human reasoning. One way to handle this is to use different time steps for the PE and IE. Several smaller values were tested with fewer calls to the IE. The results showed that there is no optimal value that is compatible with all actions. Small time steps are required to deal with some of the actions such as cutting ropes, while others, such as bursting a bubble, can be equally efficiently handled with longer time step. The alternative, and computationally more efficient, approach is to implement an adaptive time step. In this method, the PE state is still updated 60 times/sec, however fewer calls are made to the IE based on the action performed in the previous state. For example, a large time step is used after pressing an air-cushion since this action is mostly performed to change the direction of the candy and will be mostly followed by a free candy movement without any interaction with the player for a few time step. Cutting a rope on the other hand required a smaller time step since this action usually initiates another consequence actions such as cutting another rope. As a result from several game testing experiments, cutting a rope is given the lowest time step of 10, meaning that the IE will be inquired about the next actions after each 10 PE updates (6 times/sec). The void action comes next with 12 PE time steps before the next inquiry. Fifteen time steps delay is assigned for pressing an air-cushion and bursting a bubble. while rocket press is given the highest value of 17 time steps.

Implementation Details

Prolog was chosen as the first-order logic programming and reasoning engine to infer the next action to perform. A Java based implementation of the engine was used, namely JTrolog. The physics engine is implemented in C# with XNA for managing runtime environment and the two engines communicate by writing to files.

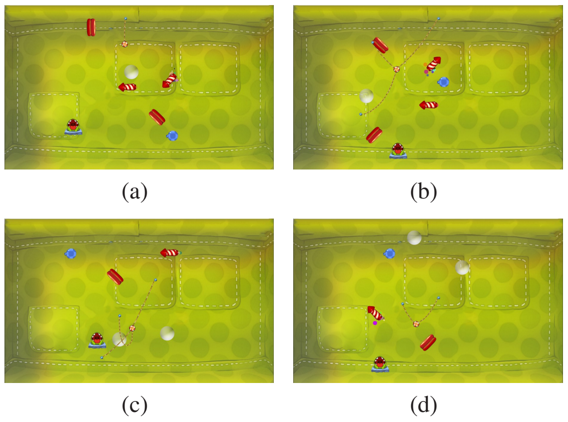


Figure 8: Selected samples of the playable levels evolved.

Experimental Setup The GE experimental parameters used are the following: 100 run of 100 generations with a population size of 20 individuals. The ramped half-and-half initialization method was used and the maximum derivation tree depth was set at 100, tournament selection of size 2, int-flip mutation with probability 0.1, one-point crossover with probability 0.7, and 3 maximum wraps were allowed.

The evolution process is terminated when a playable level is found or when the maximum number of generation is reached. The maximum depth of the state tree is limited to 80, allowing a maximum of 80 actions to be executed to test the playability of a level. This threshold corresponds to about 15 seconds of playing and was chosen after testing the model on several level structures.

6 Results and Analysis

An experiment has been conducted to evolve playable levels using the framework proposed. Evolution is repeated for 100 times starting from a random population each time. The results showed that a large number of the levels evolved did not pass the threshold defined on the design constrained presented in Section 3 (only 335 levels survived out of 4500 levels evolved). Those levels were further evolved and evaluated for playability resulting in a total of 123 playable levels. Some sample levels are presented in Figure 8.

The results showed that detecting a single playable level required an average of 29.8 ± 58.3 sec while a larger time (210.6 ± 167.6 sec) is required for detecting non-playable levels. This is because, for non-playable levels, we have to do more exploration in the state tree. The analysis of the total number of nodes explored showed that an average of 638 nodes were evaluated for non-playable levels compared to only 115 for playable designs. In order to check for the efficiency of the strategy followed by the agent, we investigated the number of branch cuts performed. The results showed that no branch cut was done in 35% of the playable levels. This means that the agent was able to solve the level by exploring the minimum number of nodes (the number of nodes explored in this case is equal to the number of actions performed to solve the level).

Analyzing the actions performed to solve the levels showed that the void action is applied in 72.3% of the game states compared to the other actions combined. This is expected since an action can be performed only once on one component (except for pressing an air-cushion) while most of the gameplay time is usually spent waiting for the candy to reach a certain position.

In order to investigate whether all components presented are actually necessary to solve the level, we calculated the number of times a component is used versus the number of time it is presented in the level for each type. Unsurprisingly, the results showed that most of ropes generates are used (note that all ropes will be attached to the candy when the game starts and they should be cut to free the candy). Most of the other components, on the other hand, were not necessary for solving the level. The results show that of all components presented in all playable levels, only 36% were used. This observation indicates that alternative approaches for evaluating the levels which might lead to more interesting level design can be considered. For example, one could design a fitness function that maximizes the number of components used. This is useful since it does not only minimize the number of unused components, but it also enhance the puzzle aspect of the design because it allows a longer, more sophisticated path to win the game.

7 Conclusions

In this paper we present an evolutionary framework for generating playable content. We combine design constraints with simulation-based playability check to evaluate each design evolved. A reasoning agent is constructed to playtest the levels following a set of rules defined in a first-order logic format imitating the reasoning approach followed by human players. Two set of rules are investigated; one that only considers components placement and information about their properties and a more advanced set that incapsulates more context knowledge in form of physics properties and trajectories. An experiment was conducted to evolve 100 playable levels. The results showed that the AI agent can efficiently detect whether a level is playable and that the overall framework can be effectively used to generate playable content.

This study opens the door for many interesting future directions in physics-based games which is a genre that has not been much explored yet in the field of procedural content generation. We are planing on investigating a number of future directions including the use of multi-objective optimization methods to generate playable yet hard to win levels. Furthermore, we are currently in an ongoing effort to make the tool accessible for human designers and players. We are working on building an authoring tool by incorporating the framework with an easy to use and to navigate user interface. The interface will provide features such as evolving complete playable levels from scratch or starting for an initial design provided by the player. It will also be possible to check if a given design is playable and to provide assistants to improve the design, by making it, for example, harder/easier to solve or give hints about the best action to perform.

References

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

Jaffe, A.; Miller, A.; Andersen, E.; Liu, Y.-E.; Karlin, A.; and Popovic, Z. 2012. Evaluating competitive game balance with restricted play. In *Proceedings 8th Artificial Intelligence and Interactive Digital Entertainment Conference*, 26–31.

Millington, I. 2007. *Game physics engine development*. Morgan Kaufmann Pub.

Ortega, J.; Shaker, N.; Togelius, J.; and Yannakakis, G. 2012. Imitating human playing styles in super mario bros. *Entertainment Computing*.

Shaker, M.; Sarhan, M.; Al Naameh, O.; Shaker, N.; and Togelius, J. 2013. Automatic generation and analysis of physics-based puzzle games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):187–200.

Smith, G.; Whitehead, J.; and Mateas, M. 2010. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216. ACM.

Togelius, J., and Schmidhuber, J. 2008. An experiment in automatic game design. In *IEEE Symposium On Computational Intelligence and Games. CIG'08*, 111–118. IEEE.

Togelius, J.; Preuss, M.; Beume, N.; Wessing, S.; Hagelbäck, J.; and Yannakakis, G. 2010a. Multiobjective exploration of the starcraft map space. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 265–272. Citeseer.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2010b. Search-based procedural content generation. In *Proceedings of EvoApplications*, volume 6024. Springer LNCS.

Yannakakis, G. 2012. Game ai revisited. In *Proceedings of the 9th conference on Computing Frontiers*, 285–292. ACM.