Procedural Puzzle Generation: A Survey

Barbara De Kegel and Mads Haahr, *Member, IEEE*

***Abstract*—Procedural content generation (PCG) for games has existed since the 1980s and is becoming increasingly important for creating game worlds, backstory, and characters across many genres, in particular, open-world games, such as *Minecraft* (2011) and *No Man's Sky* (2016). A particular challenge faced by such games is that the content and/or gameplay may become repetitive. Puzzles constitute an effective technique for improving gameplay by offering players interesting problems to solve, but the use of PCG for generating puzzles has been limited compared with its use for other game elements, and efforts have focused mainly on games that are strictly puzzle games, rather than creating puzzles to be incorporated into other genres. Nevertheless, a significant body of work exists, which allows puzzles of different types to be generated algorithmically, and there is scope for much more research into this area. This paper presents a detailed survey of existing work in PCG for puzzles, reviewing 32 methods within 11 categories of puzzles. For the purpose of analysis, this paper identifies a total of seven salient characteristics related to the methods, which are used to show commonalities and differences between techniques and to chart promising areas for future research.**

***Index Terms*—Procedural content, puzzle games.**

I. Introduction

PROCEDURAL content generation (PCG) has been popular in digital games for decades; from dungeons and levels in *Rogue* (1980) to terrain and resources in *Civilization* (1991) and *Minecraft* (2011), and most recently, entire planets and solar systems in *No Man's Sky* (2016), it has often been used in the creation of game worlds across genres. Besides worldbuilding, PCG has also been used in the creation of narrative elements. For example, *Dwarf Fortress* (2006), a predecessor and influence on *Minecraft* [1] procedurally generate a detailed backstory including a dwarven lineage at the start of each game. Characters driven by artificial intelligence (AI) have also been the subject of PCG, such as *Crusader Kings II* (2012), which procedurally generates character traits that alter the decision-making of the non-player characters (NPCs), decision making, leading to interesting family dramas, whereas *Shadow of Mordor* (2014) makes players feel like they are fighting specific enemies by generating each enemy's personality independently and retaining it throughout the game.

Manuscript received January 12, 2017; revised October 31, 2017 and September 20, 2018; accepted May 9, 2019. Date of publication May 20, 2019; date of current version March 17, 2020. *(Corresponding author: Mads Haahr.)*

B. De Kegel was with the School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland. She is now with Systems Biology Ireland, University College Dublin, Dublin, Ireland (e-mail: barbaradekegel@gmail.com).

M. Haahr is with the School of Computer Science and Statistics, Trinity College Dublin, Dublin, Ireland (e-mail: mads.haahr@tcd.ie).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TG.2019.2917792

The benefits of PCG are obvious—by replacing the problem of creating game worlds, backstory, and characters with the metaproblem of creating systems that in turn create these things, content production costs can be reduced. At the same time, there is potential that the content can be scaled in a near-unlimited fashion. A particular challenge in the construction of game worlds, backstory, and characters in this fashion is of course that the content generated with PCG may become repetitive and uninteresting. Filling large open worlds with engaging content is challenging, and this is particularly true for games with procedurally generated worlds, such as *Minecraft* and *No Man's Sky*. While *Minecraft* has successfully profiled itself as a crafting sandbox, *No Man's Sky* has been criticized by players for being boring in the moment-to-moment gameplay, leading many to ask for refunds [2].

A. Why Procedurally Generate Puzzles?

While PCG is popular for generating many types of game content, including game worlds, entity behaviors, and characters [3], its use in the creation of puzzles has been limited. Efforts have focused mainly on puzzles for games that are strictly puzzle games, rather than creating puzzles to be incorporated into other genres, such as role-playing games (RPGs), where they can improve gameplay by offering players interesting problems to solve.

We define puzzles as problems to which the player can find a solution based on previous knowledge (from in or outside the game) and/or by exploring the solution space [4]. It is clearly interesting to generate puzzles algorithmically for the simple reason that they are popular with players but that the replay value of individual puzzles is low.

In digital games, puzzles are a feature of many game genres and come in a myriad of forms. Sometimes they are integrated into game environments, such as the physics puzzles in *Half-Life 2* (2004), and other times they are present as minigames that serve as intermissions from the main gameplay, such as the hacking puzzles in *Bioshock* (2007). Puzzles in digital games are popular when they offer players interesting problems to solve, and they work particularly well when they integrate with the core gameplay.

Procedural generation of puzzles comes with similar challenges as procedural generation of other game elements. A barrier for mainstream adoption of generated puzzles, compared to other types of content, may be the strict solvability constraint. As noted previously, large procedurally generated game worlds often suffer from repetitive content, and for puzzles, this repetition can manifest as a single type of interaction or puzzle appearing in many different places, e.g., the bypass puzzles in *Mass Effect 2*

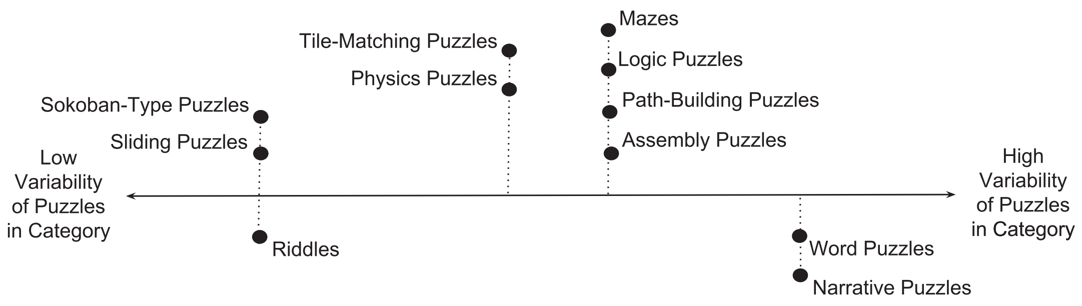


Fig. 1. Map of puzzle categories. Categories toward the left-hand side are fine-grained, containing highly similar puzzles, whereas categories toward the right-hand side are coarse-grained, containing more dissimilar puzzles. Vertical positioning is not significant.

(2010). The outcome of applying PCG is usually unpredictable, which can be a deterrent, but also a benefit, especially insofar that it can aid designer creativity.

In the context of smaller, story-driven games, puzzle generation can be applied toward improving replayability; a common complaint for these types of games is that they can only really be played once. Changing the puzzles integrated into the game's story could provide variety in experience without the need for designers to write branching story lines.

B. Review Methodology

Previous surveys of PCG for games, covered in [3] and [5], look at generation techniques used for a wide range of game content—covering everything from 3-D objects to game systems. Puzzles are one type of game content addressed in that range, but, given the broadness of these surveys, only a limited view is provided. Our survey aims to focus specifically on research in PCG for puzzles.

Work on procedural puzzle generation has mostly focused on specific puzzles, because creating a generator requires at least some knowledge of specific puzzle rules. Our approach in this paper is to review research in puzzle generation by puzzle types; we divide the space of all puzzles that have been subject to PCG into categories that reflect common puzzle genres. The categories are not based on a formal survey of puzzles and are not intended as a taxonomy of puzzles. Instead, they are based on our informed reading of how the different puzzle genres are discussed in the literature and as an approximate grouping of related puzzles that will allow us to review a considerable body of work in a methodical manner. Effectively, we divide a wide domain and look for relationships between PCG techniques and puzzle types. Puzzles' characteristics often carry implications for the possibilities and challenges of PCG. To that end, we place the category boundaries where we determined the puzzle characteristics were different enough to influence PCG.

Because the categories are approximations, it is not always clear-cut which category a given puzzle belongs to, and we will give examples of this later in the paper. Furthermore, the puzzle categories are not intended to be exclusive, i.e., there are puzzles that fall in the intersection of two (or more) categories. The categories are also not intended to be all-encompassing; in particular, we do not include categories for puzzles with little associated PCG work, such as time-based puzzles, such as those found in *Braid* (2008) and *The Talos Principle* (2014) or programming puzzles, such as those found in *Lightbot* (2008).

For the same reason, the categories also differ in granularity. Some categories primarily contain puzzles that are very similar, whereas others contain puzzles that differ significantly, even if their core idea is the same. We have tried to show this in Fig. 1 where the 11 puzzle categories are arranged broadly according to the variability of the puzzles in that category. Our estimate as captured in the figure should not be considered a formal metric, but rather an informed assessment based on our reading of the field. We leave the exhaustive mapping of the domain of puzzles themselves as future work and instead focus this paper on PCG techniques.

Not every puzzle type is weighted equally in this survey because generation is trivial for some types of puzzles, and not feasible for others. We determined it worthwhile to include categories with little work in them as they help provide a more complete picture of the map as a whole.

For our literature search, we searched for works that were both frequently referenced and those that were somewhat obscure in order to cover what was influential as well as what has been attempted outside of the mainstream research. We generally focused on academic and experimental works as opposed to published games.

For each puzzle type, we give a brief introduction to the puzzle characteristics, list some prominent examples of games that include or consist of puzzles that we deem belong in that category, and then we review the work that addresses the procedural generation of these puzzles. This approach is intended to group research that has similar objectives and at the same time allows us to make observations across the puzzle types. A further intention is that this approach will allow us to identify "gaps on the map" of research into puzzle generation, i.e., areas that are underdeveloped and, therefore, promising candidates for future research.

We define eight salient characteristics of PCG methods based on the taxonomy devised by Togelius *et al.* [6] in their survey of search-based PCG , which was later revised by Shaker *et al.* [5] in their book on PCG in games.

Our selection of characteristics is tailored to puzzle generation, so, we have left out some dimensions and added others. Specifically, we do not include the "necessary versus optional" dimension, because all the work surveyed generates puzzles that constitute an entire game, i.e., the puzzles generated are always necessary, and “stochastic versus deterministic,” because deterministic methods are not very interesting for puzzles. The dimensions we have added are “direct versus simulation-based evaluation” (see Section I-B2) and “quality considerations” (see Section I-B7).

The purpose of these characteristics is primarily to provide a basis for analysis of PCG methods currently used for puzzle generation. Similarly to the dimensions listed by Shaker *et al.* in their book, the items on the list below should be seen more as a spectrum than a dichotomy. It should also be noted that the list does not provide an exhaustive means of comparison, and we expect it to change as the field evolves.

1) Constructive Versus Generate-and-Test: Constructive algorithms generate the content once and are done, usually performing validity checks at different stages of construction. The Markov chain is a typical example of a constructive algorithm. Generate-and-test techniques construct and test in a loop until a satisfactory candidate is found; here, evaluation occurs each time a complete candidate has been constructed. Some search-based algorithms, including answer set programming solvers, used by various generators in our survey, fall somewhere between constructive and generate-and-test algorithms. A search often creates, tests, and rejects partial or potential candidates before they are fully generated [7].

2) Direct Versus Simulation-Based Evaluation: In direct evaluation, the fitness score of a candidate is based on features that can be extracted directly from the generated content, e.g., the level layout. In simulated-based evaluation, an AI agent attempts to solve the puzzle, implying that automatic solvers are sometimes used in puzzle generation. There is also a third, less common, kind of evaluation—interactive—where a human implicitly or explicitly scores candidates [8]. Evaluation functions are necessary for any algorithm that has some form of generate-and-test loop and are often not relevant for constructive algorithms, such as constraint-based methods, because those formulate the suitability of puzzles as a property of the allowed solutions.

3) Online Versus Offline: Online generation takes places while the game is running, allowing the player to see a lot of content variation, whereas offline generation occurs during game development, or at the start of the game. An algorithm must be both sufficiently fast and reliably accurate to be suitable for online generation.

4) Degree and Dimension of Control: Designers can control which content is generated by a PCG technique in different ways. Shaker *et al.* mention the random seed and a vector of content features as examples. In some cases, this control is in the form of optional tweaking of the generator but it can also refer to the amount of input a game designer has to feed into the puzzle generator. For generators that require a lot of input, the quality of the resulting puzzles tends to be heavily dependent on the quality of the input. While an input requirement allows for creative control, it also means that the generator will not work out of the box. Examples of input are templates of level layout pieces or databases of game item specifications.

5) Automatic Generation Versus Mixed Authorship: Mixed authorship is closely linked to the previous characteristic; it refers to content that is generated as the result of cooperation between the game designer and the generation algorithm. However, while the dimension of control is always about input that the designer feeds to the algorithm, mixed authorship may also refer to designers continuing the work of the generator, i.e., completing the design of a partially generated puzzle. Mixed authorship necessarily implies the algorithm running offline.

6) Generic Versus Adaptive: The output of generic algorithms is independent of the individual player, and this is the category most generators fall under. However, there is some investigation into adaptive PCG, which uses the player’s actions as inputs to the generator. While the use of PCG is often cited as useful for creating replayability, PCG can also be used to introduce adaptiveness [9].

7) Quality Considerations: Puzzle generators that include quality constraints/specifications attempt to generate puzzles that are also interesting and appealing, in addition to being simply solvable. Different methods may devote varying amounts of resources toward puzzle quality. One aspect of quality is difficulty, i.e., whether the puzzle generation algorithm allows for the creation of puzzles of different, specified levels of difficulty.

C. Traditional Versus Digital Puzzles

Some puzzle types covered in this survey originated on paper or in other tangible formats, whereas others, such as physics-based puzzles, can only really exist in digital games, due to their interactive and dynamic nature. Dynamic puzzles present unique challenges for generation, as will be discussed further in this paper. Traditional puzzles are often digitalized as is, i.e., they are adapted to a digital platform without changes to the core puzzle mechanics. Popular examples of such puzzles are *Sudoku* and crossword puzzles. Digitalization can make traditional puzzles more available, e.g., on smart phones, and accessible, by providing ways to easily undo and check partial solutions.

One notable difference between traditional and digital puzzles is that the latter can employ brute force as a game mechanic. For example, for each set of maze puzzles in *The Witness* (2016), the puzzler has to figure out the rules by using brute force to solve several small puzzles. Although the digital context makes brute force possible as a game mechanic, there are also computer-based puzzles that do not lend themselves to this, such as the path-building puzzles in *The Talos Principle*.

Another difference between paper-based and digital puzzles is the existence of temporal aspects; digital puzzles more frequently use time as a mechanism or constraint than traditional puzzles, where it is only common in competitions. Time pressure can be used to increase the difficulty of a puzzle, e.g., *Tetris* (1984), an interactive puzzle, would not be challenging without its temporal component.

Distinctly from games with a time dimension, there are games that use time as a puzzle mechanic. The solutions of these puzzles require player-directed manipulation of time, e.g., the puzzles in *Braid*.

Spatial thinking is required for a broad set of puzzles in different media. Interlocking puzzles, such as the assembly of a cube out of irregular pieces, are an example of nondigital 3-D puzzles based on spatial thinking. In digital games, a puzzler would use

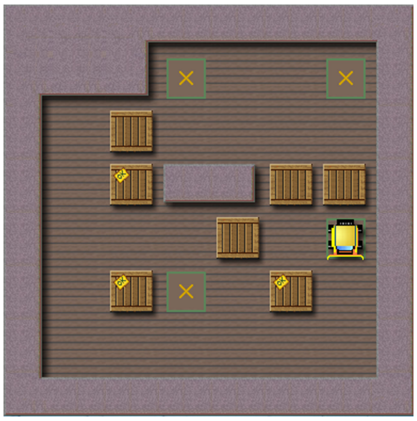


Fig. 2. Screenshot from JSoko, an open source version of *Sokoban*.

spatial thinking to, for example, conceive of logical placements of items in space to form a solution. This description applies to both 2-D and 3-D path-building puzzles, e.g., *Refraction* (2010) [10].

II. Survey of Procedural Puzzle Generation

A. *Sokoban-Type Puzzles*

This category of puzzles is named after the 1982 Japanese video game in which a player character pushes crates around a constrained grid-based area to get them to goal positions. A defining factor of this type of puzzles is that no items/characters are ever lost or added to the board; the solution exists as a rearrangement of the original configuration. Many variations and offshoots of *Sokoban* (1982) now exist, a level from one version, JSoko, is shown in Fig. 2. Game developer Stephen Lavelle, creator of *Stephen's Sausage Roll* (2016), a popular take on *Sokoban*, has even created a scripting language for this type of game, called *PuzzleScript* [11], which has been used as a tool for automatic generation and evaluation of Sokoban-style puzzles [12], [13].

The constraints imposed by limited space and possible actions, e.g., crates can be only be pushed, not pulled, are a key characteristic of *Sokoban* puzzles. The player has to think a few moves ahead, as some sequences of moves may lead to a state from which the solution is unreachable. In practice, *Sokoban* puzzles often have an undo functionality that allows for avoiding that scenario. This facilitates the puzzler's problem-solving, as it allows exploration of solution sequences, and means the use of brute force is technically possible. However, all but elementary *Sokoban* puzzles tend to have several open lines of play at any given time, making brute force unfeasible.

Well-designed *Sokoban* puzzles must strike the balance between trivially easy and outright impossible, a difficult task for all but experienced *Sokoban* level designers [14]. Procedural generation can function as an aid to puzzle designers. Generally *Sokoban* levels do not have many, if any, alternative solutions. The problem-solving process involves using spatial thinking to pursue promising sequences, and discarding those that are futile.

Despite the simplicity of the rules, *Sokoban* puzzles can be challenging to solve [15], for both human and machine players. Past research has determined that solving generalized *Sokoban* puzzles, i.e., on an $n \times n$ board, is PSPACE-complete [16]. Automatic puzzle solvers are not the focus of this paper, but are relevant insofar that they are often used to test the solvability of a generated puzzle.

Sokoban-type puzzles are relatively popular in the domain of procedural puzzle generation, possible due to the fact that *Sokoban* exhibits compelling challenges in this field, including a large space of possible configurations, which hinders exhaustive search algorithms and may make it difficult to guarantee solvability [15]. Additionally, there is no good method to evaluate if an initial state will lead to a nontrivial or interesting solution sequence.

One of the earliest forays into puzzle generation was by Murase *et al.* [17] who developed a program to create *Sokoban* problems in three stages; generation, checking, and evaluation. This is a generate-and-test approach: a level layout is generated through a random combination of level templates and random placement of game items, and then a breadth-first search (BFS) solver is used to check for a solution. However, BFS will only manage to solve puzzles with short solution sequences, so those with long sequences were incorrectly discarded [17]. After testing for solvability, trivial, and uninteresting, albeit legal, levels are discarded at the hand of an evaluator that checks solution length, number of direction changes, and number of detours. One of the main issues with this generation program is that the restriction on solution length prevents the creation of complex problems. This method can be controlled through the use of the level templates, and is not suitable for online generation because the BFS results are unpredictable.

Taylor and Parberry [14] generated *Sokoban* levels that are guaranteed to be solvable on the basis of working backward from the goal positions. As in the approach by Murase *et al.*, empty rooms are generated with templates, but here, invalid or low-quality rooms are immediately discarded. A brute force algorithm is used to evaluate all possible combinations of goal positions. While this can lead to the discovery of compelling levels, it is an expensive process; the runtime of the algorithm is exponential. For each goal placement, the system finds the furthest possible starting position, i.e., the shortest longest path according to the box line metric, by moving in reverse [14].

Taylor and Parberry claim their technique produces interesting levels, but it is only suitable for offline generation because it runs in exponential time and cannot handle levels with more than six boxes. The authors mention that their methodology could be applied for generation of other puzzles, but there would be quite some effort involved in reducing the amount of game-specific information; as mentioned, a major issue in puzzle generation efforts.

Taylor *et al.* [18] performed an auditory Stroop test that indicated that players are as engaged with the generated puzzles by Taylor and Parberry [14] as they are with hand-crafted puzzles by experienced designers. The experiment exploits the fact that attention is a finite resource; focusing on *Sokoban* will decrease participants attention on the Stroop test and vice versa. That players found generated puzzles equally interesting demonstrate value in procedurally generated puzzles.

Recently, Kartal *et al.* [15] have worked on procedurally generating *Sokoban* levels of varying sizes and difficulty using a Monte Carlo tree search (MCTS) approach. They apply MCTS by defining the puzzle generation as an MCTS optimization problem. This approach has been successful for other problems with high branching factors, the search tree structure guarantees solvability, and it has the anytime property, meaning the algorithm will return a valid solution regardless of when it is interrupted. The best puzzles found after different roll-outs can be stored; the search could optionally be terminated at a certain quality threshold. The authors state that their method is able to generate puzzles quickly enough for them to be used in procedurally generated minigames.

The puzzle generation method by Kartal *et al.* [15] splits up the creation of the initial room layout and the placement of the goal locations. Initially the board is composed entirely of obstacles except for one empty tile with an agent, and the possible actions that can be undertaken at each node in the search tree are: "delete obstacles," "place boxes," or "freeze level." Once the "freeze level" action is chosen, saving a start configuration, the action set is replaced by the "move agent" and "evaluate level" actions. The "move agent" action simulates *Sokoban* game play; the boxes are pushed around to determine the goal positions. Like Taylor and Parberry, Kartal *et al.* exploit the fact that generating a puzzle through game play guarantees solvability, but they execute game moves in the forward direction, whereas Taylor and Parberry went backward.

MCTS requires an evaluation function to guide the search. Kartal *et al.* have published two different approaches for this function, both are direct but one is theory driven, whereas the other is data driven. The theory-driven approach uses a combination of two metrics based on the level layout: terrain, the number of neighboring obstacles, and congestion, the number of obstacles and items between each box and its corresponding goal [15]. While this method eliminates the need for human input, the evaluation function did not capture all aspects of difficult *Sokoban* puzzles, and has no formal validation.

For the data-driven approach, Kartal *et al.* performed a user study to annotate a set of existing *Sokoban* puzzles with perceived difficulty. Statistical analysis was performed on the results to discover which features correlate most strongly with difficulty, and these were then used in the evaluation function. Features were restricted to those that are efficient to compute, in order to maintain the efficiency of this method. Each given run of the MCTS algorithm will generate several levels of increasing difficulty. A second user study was performed for validation; there was a high correlation between the score assigned by the MCTS and the perceived difficulty [19].

There is still future work to be done on generating large *Sokoban* puzzles of high difficulty; for all described methods, generation time increases exponentially for linear increases in the number of boxes and tiles.

Fling! (2013), shown in Fig. 3, is an example of a puzzle that falls in the overlap of *Sokoban*-type and tile-matching puzzles, though more toward the former. In this game, the puzzler flings balls into each other to sequentially remove all but one ball from an empty grid. The balls act in turn as the player character, an obstacle, or a crate, when compared with *Sokoban*.

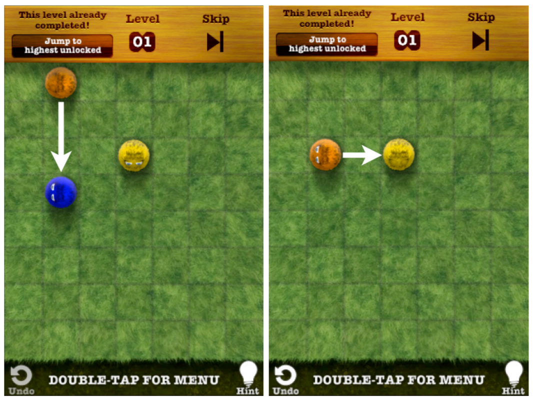


Fig. 3. Sample moves on a *Fling!* (2013) board, from [20].

Sturtevant [20] looked at using large-scale BFS—a complete and uninformed search approach—for analysis and content generation for *Fling!*. This is unusual as PCG methods generally favor selective search techniques due to the large state space. Sturtevant attempts to answer a question posed in [6], namely whether search-based PCG can be combined with a top–down approach.

The focus of his research is the development of a tool for designers that can analyze and explore *Fling!* puzzles, rather than generating them from scratch, although it can be used for this also. The tool uses an endgame database, generated by solving all *Fling!* boards of sizes 1–10 using a retrograde search. For any given board, the tool can determine the following metrics: the number of states legally reachable, using forward BFS; the legal moves that lead to a goal state, using depth-first search (DFS) with endgame data; and how adding/removing pieces from the board changes the solvability. As such, the tool could be used in a generate-and-test puzzle generation approach.

The difficulty of a given *Fling!* board is most intuitively measured by the number of reachable states from an initial configuration. Experiments showed a strong correlation between levels (difficulty) and number of states in the state space. Like for *Sokoban*, more domain-specific metrics may be useful, e.g., counting the number of times the player has to switch which ball is being flung.

B. Sliding Puzzles

Sliding puzzles are closely related to *Sokoban*-style puzzles because they also involve moving items, or tiles, toward goal positions in a constrained grid-based space. There are, however, some differentiating characteristics that qualify this as a different puzzle type: there is no player character and items can be moved in any free direction. Often, the grid is square-shaped without obstacles and there are only a few open spaces to slide a tile onto. Like for *Sokoban* puzzles, the player must look a few moves ahead to determine possible winning sequences.

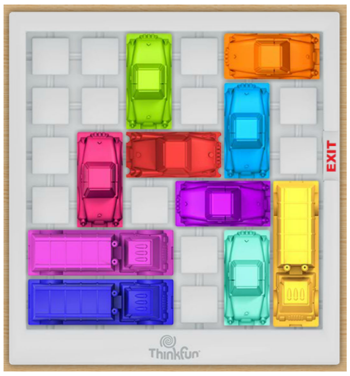


Fig. 4. Screenshot from *Rush Hour* (1996).

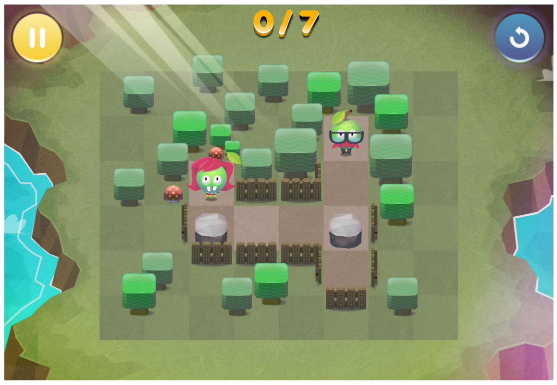


Fig. 5. Screenshot from *Fruit Dating*.

The most well-known examples of sliding puzzles are *Rush Hour* (1996), the 15-puzzle, and the picture-forming sliding puzzles, which all existed in a nondigital format first. There are generally no unrecoverable states, as all moves are reversible, which encourages exploration of the solution space, possibly using brute force. Unlike in *Sokoban*-type puzzles, sliding puzzles may have one-to-one pairings between items and goal positions, in which case, figuring out the best mapping is not part of the solving process. In *Rush Hour* and similar puzzles, there is one item that must reach one defined goal position—in *Rush Hour*, shown in Fig. 4, it is the red car that must reach the exit. Like *Sokoban*, the *Rush Hour* puzzle has been shown to be PSPACE-complete [21], and exhibits similar challenges in determining the difficulty of a given puzzle.

Block-sliding puzzles can conceivably be generated by starting from the end configuration and working backward, in a similar fashion to some of the generation methods described in Section II-A. However, the number of moves used to play backward to a start layout could be greater than the shortest path, so it is not necessarily a good metric for difficulty.

Some work has been done on the generation of hard configurations of the *Rush Hour* puzzle game [22], [23]. They define a hard puzzle as one requiring a large number of moves to solve, noting that a puzzle with many moves will still be easy if there are always only a few possible moves. The researchers used symbolic methods to iteratively compute reachable configurations from a set of solvable initial configurations for *Rush Hour*. This implies a constructive method. Symbolic methods are a way to bypass combinatorial explosion in many typical applications: sets are represented symbolically, in this case, through use of binary decision diagrams (BDDs) instead of element-wise. This allowed for studying the huge graph of all possible initial configurations of the 6×6 *Rush Hour* puzzle. A dual encoding of the board, i.e., one that works on a line and column level, is used to limit the size of the BDDs. The hardest initial configuration, found by Collette *et al.* takes 93 steps to solve. Additionally, there are 24 132 configurations that can be reached from it. As part of discovering the hardest configuration, the researchers classified every solvable configuration according to minimal solution length.

C. Tile-Matching Puzzles

The player's objective in tile-matching games is to manipulate tiles on a grid in order to make matches [24]. When a match is made, the corresponding set of tiles disappears and the player scores points. Common matching criteria include shapes, colors, and symbols. Puzzles of this category are relatively simple—they have very few rules—and are often categorized as casual games. Unlike other types of puzzles, tile-matching puzzles are almost exclusively randomly generated, and generally do not have one specific solution.

This history of tile-matching puzzles could be traced back to *Tetris*, which falls in the overlap between the tile-matching and packing puzzle categories. Like in *Tetris*, each player action in a tile-matching puzzle permanently changes the level layout.

The most popular subcategory of the tile-matching game is match-three games, e.g., *Bejeweled* (2001), in which players swap the positions of tiles to make a row or column of at least three matching tiles. Juul [24] observes that this category of puzzles has a "low status," perhaps due to their low barrier to entry, or the large number of similar games that now exist. Most tile-matching puzzles have an element of time pressure that introduces a fail state; without a timer, the puzzles would be too easy and the puzzlers would not feel a sense of achievement.

The initial layout of tiles on the grid and/or the choice of tiles/pieces that appear during the game are likely always procedurally generated using random number generators. The generation of these simple tile-matching puzzles is a rather trivial task, but for hybrid games, more interesting approaches exist.

Rychnovsky procedurally generated all the levels for his game *Fruit Dating* (2014) [25], a more complex tile-matching puzzle that draws elements from *Sokoban* and sliding puzzles, and has similar mechanics to the popular mobile puzzle game *Threes* (2014). *Fruit Dating*, shown in Fig. 5, consists of moving items on a grid-based board with walls and obstacles, with the goal of getting matching characters onto adjacent tiles. Unlike *Sokoban*, it lacks a player character, and each action can affect between zero and all items on the board. Items are moved by swiping in one of the four cardinal directions; each swipe will move all objects in the chosen direction while they are not blocked by obstacles.

As is characteristic for the *Sokoban* and sliding puzzle types, *Fruit Dating* requires the puzzler to rearrange items while anticipating the outcome of each move to prevent dead ends. A matched pair is eliminated, and there are a few items on the board with special abilities, which are common traits of tile-matching games. The challenge for *Fruit Dating* stems from the puzzler having to discover a specific sequence of moves; this game has well-defined solutions, whereas typical tile-matching games tend to be more open-ended.

Rychnovsky developed a level editor tool that can generate new levels and evaluate the solvability of any given level. The automatic solver uses a BFS approach with pruning and returns the shortest path. The level editor interface means that the designer can tweak a generated level, and then retest it to make sure it is still solvable.

The process of generating a *Fruit Dating* level is divided into two steps: generating the level structure and placing the on-board items [25]. The level structure is created by first randomly placing the external wall tiles, and then the internal ones. The game objects, including the pair(s) of fruits that must be matched, are placed randomly according to predefined weights assigned to empty tiles. Rychnovsky took aesthetic qualities into account when creating the layout in order to facilitate the creation of interesting looking levels.

All levels are checked for solvability after generation, and those that are not solvable are simply discarded. The main issue with this generation approach is that there is no way to control difficulty, other than by simply adding more items to the board. Overall, the output is not of high quality, and the approach functions more to aid rather than to replace the designer.

D. Assembly Puzzles

In an assembly puzzle, a number of shapes must be assembled into a larger shape without overlap or gaps. Generally, all the provided shapes must be used to achieve this. The process of fitting the pieces into the larger shape may be entirely based on shape-matching, or it could involve creating a picture out of the images printed on the pieces, as is the case for jigsaw puzzles. Puzzles of this type could exist in either a 2-D or 3-D format, and may or may not be digital, though the 3-D versions tend to tangible. A famous 3-D assembly puzzle, also called an interlocking puzzle, is the Soma cube, invented by Piet Hein in 1936, which consists of seven irregular pieces made up of unit cubes that must be assembled into a cube with side length three.

In *The Talos Principle*, unlocking new areas requires solving an assembly puzzle with tetromino pieces. As mentioned earlier, we consider *Tetris* to be a combination of an assembly and tile-matching puzzle; the goal there is to assemble pieces into rectangles without overlap, and matching tiles into a line shape will cause them to disappear. *Tetris* is set apart from other assembly puzzles by its time pressure element and the inability of the player to move pieces once they have been placed. Time pressure is a label that can be applied across puzzle types, rather than being a type on its own.

A famous example of an assembly puzzle is the *Eternity* puzzle created by Christopher Monckton in 1999. Composed of assembling 209 irregular polygons of the same color into a dodecagon, this puzzle is extremely difficult and came with a £1 million prize for whoever could solve it in the first four years. The prize was awarded to two Cambridge mathematicians in October 2000.

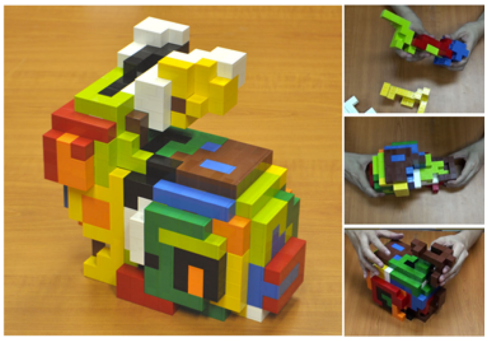


Fig. 6. Example interlocking puzzle with LEGO bricks, from [26].

There is no formal research into the generation of 2-D assembly puzzles, likely because they are relatively uncomplicated to create. Conceivably, such a puzzle could be generated by an algorithm that splits a region into areas, e.g., a Voronoi diagram.

Song *et al.* [26] developed a constructive approach for generating interlocking puzzles, such puzzles are made up of a number of pieces that can be assembled into an structure that can be locked by a single key piece. They are challenging to solve and compute as the pieces have no orientation and can be combined in an extremely large number of ways. In a standard interlocking puzzle, all pieces are immobilized until the key piece is removed, at which point they all become mobilized. However, in a recursive interlocking puzzle, which was the specific focus of the research, pieces are also locked in intermediate states.

A model for generating new interlocking geometric structures, given a general voxelized shape as input, was derived from the analysis of interlocking mechanics. Song *et al.* [26] prove that the whole model can be interlocking if every three consecutive pieces are interlocking. A puzzle is created by iteratively extracting pieces from the given shape, in such a way that validity is guaranteed. This technique provides a designer with a lot of control as it can create puzzles from any voxelized shape, enabling the creation of, for example, custom puzzles that use LEGO bricks, as shown in Fig. 6.

E. Mazes

We define mazes as puzzles that require the puzzler to find a valid path from a starting point (entry) to an ending point (exit). A huge variety of puzzles could be created from rules for safe movement in an otherwise hazardous environment [27]. These puzzles could have explicit barriers, such as in most traditional mazes, where the path is obstructed by physical walls, or implicit boundaries, such as in some grid-based puzzles, which are sometimes referred to as logic mazes.

Many of the puzzles from *The Witness* fall under the second category; the player has to traverse a path through a grid according to some logical rules. For example, Fig. 7 shows two

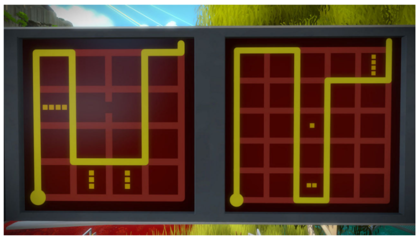


Fig. 7. Screenshot of two maze puzzles from *The Witness*.

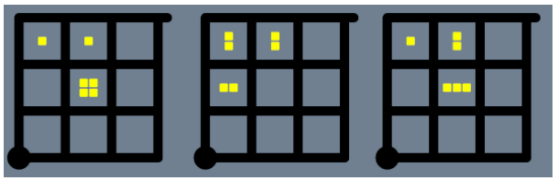


Fig. 8. Puzzles based on those from *The Witness* that were generated by Sturtevant and Ota. The correct solution is the one that works for all three puzzles.

solved implicit mazes for which the correct path corresponds to an outline that can incorporate an arrangement of all tetronimo shapes. The puzzles, and puzzle progression, in *The Witness* are carefully designed, e.g., some puzzles will "break" the heuristics, the player may have previously learned from other puzzles. This sort of design is difficult for a machine, but Sturtevant and Ota [28] have worked on an algorithm for generating a version of the tetronimo maze puzzle from *The Witness* where multiple puzzle panels must be solved jointly. They use a branch and bound (semi-exhaustive) search to enumerate possibilities and then select subsets where there is only one solution that works for all puzzles in the subset. Fig. 8 shows one such subset [28].

Obstacle course navigation can be classified as a type of maze; the player must navigate an area along a correct path to reach an endpoint without taking too much damage. Many obstacle courses are a cross between a maze and a path-building puzzle, where a player must utilize a number of items to create the desired path between entry and exit. *Loderunner* (1983) and *Lemmings* (1991) also fall into that overlap of puzzle types; they both include dynamic mazes that the player must navigate by making decisions in real-time. As such, they can also be viewed as early real-time strategy (RTS) games, and do not always have a clear-cut solution.

Constructive maze generation is an old topic in computer science, mainly popularized in the context of dungeon and/or level generation. A practical approach to random maze generation, which could be used for a number of applications and which covers classic algorithms, such as the binary tree algorithm and Kruskal's algorithm, is given in Buck's book *Mazes for Programmers: Code Your Own Twisty Little Passages* [29]. Shaker *et al.* [30] describe four main categories of constructive maze generation: space partitioning, which performs (usually binary) spatial subdivisions in a hierarchical fashion; agent-based algorithms; cellular automata; and algorithms based on generative grammars. The last two categories are also delineated by van der Linden *et al.* [31] in their survey on procedural generation of dungeons, which additionally includes sections on evolutionary and constraint-based algorithms. Nelson and Smith cover

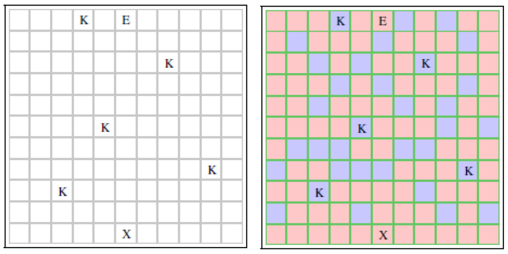


Fig. 9. Example of a chess maze for a rook from [27]. E represents the entry tile, X the exit tile, and the K tiles have knights on them. The left-hand side image is what appears to the player and the right-hand side image has the obstructed squares marked in blue.

this last category in their application of answer set programming (ASP) to mazes (ASP is described later in this section); they represent mazes as grid-embedded trees and impose constraints on reachability and path length [32]. Recently, Brewer conducted a general audience survey on the history of dungeon generation, ranging from *Rogue* (1980) to *Minecraft* (2011) [33]. We will not go into more depth on these algorithms, because most of the literature is about generating level structures as opposed to stand-alone puzzles and, thus, not within the scope of this survey. However, some work in this domain, particularly under grammar-based algorithms, focuses on the gameplay, i.e., narrative progression, aspect of dungeons, and we will look at this work in the narrative puzzles section.

Ashlock uses an evolutionary algorithm to generate chess-based and chromatic mazes of varying levels of difficulty [27]. Evolutionary algorithms are well suited to creating a large collection of unique puzzles because they can quickly locate many diverse optima in a complex function. The fitness function, which returns a measure of the quality and/or difficulty of puzzles in a generation, uses dynamic programming to calculate the minimum number of steps required to traverse the maze. Dynamic programming is a method for solving a complex problem by decomposing it into simpler subproblems, and storing the solutions to those problems.

Both the chess and chromatic mazes are grid-based and have implicit barriers; players must figure out which tiles are safe according to the puzzle rules. Entry and exit squares are marked on the grids. In the chess maze, shown in Fig. 9, the player must move across the board according to their assigned chess piece and without traversing any squares covered by the other pieces. In the chromatic puzzle, a safe move consists of continuing onto a tile whose color is adjacent in the color wheel to the color of the current tile. In digital format, some form of repercussion can be used to prevent the use of brute force.

To determine the fitness of a maze, dynamic programming is used to traverse a network and record the cost of arriving at each node, computing the shortest path to traverse the maze. During evolution, the aim is to maximize the length of this shortest path, theoretically increasing the difficulty of the maze. The evolutionary algorithm is straightforward; each generation, seven population members are randomly selected, and the two most fit out of those are picked for reproduction, which includes crossover and mutation operations [27].

Experiments had a zero rate of duplication of solutions, showing the diversity that can be achieved with an evolutionary

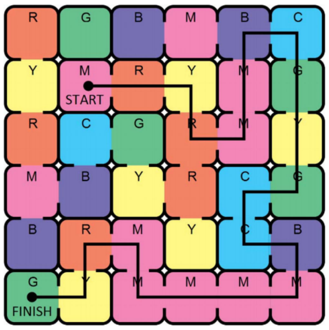
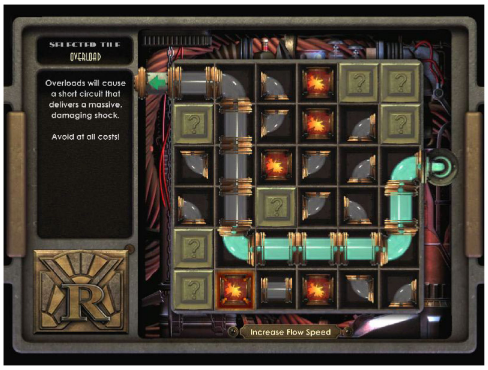


Fig. 10. Example of a chromatic maze from [7], with a shortest solution path.



Overloads will cause a short circuit that delivers a massive, damaging shock.

Avoid at all costs!

Fig. 11. Screenshot from *BioShock* (2007).

algorithm. The size of the sample space aids in this, but for smaller sample spaces, diversity-promoting measures could easily be introduced. By adjusting the dynamic programming code, the technique described could be used for other types of puzzles. However, the authors warn that dynamic programming is not a very human way of solving a maze, so it may not provide the best estimation of what a puzzler would find difficult [27].

Smith and Mateas reimplemented Ashlock's chromatic and chess maze generators using answer set programming [7]. ASP is a form of declarative logic programming where constraints and logical relations are declared in a Prolog-like language; specifically, common answer set solvers use AnsProlog [34]. Answer set programs contain two structures—facts and rules—which capture the design space model. Facts are statements that describe configurations and properties; rules control the production of new facts. There are three types of rules: choice rules, deductive rules, and integrity constraints. An answer set solver often works by performing a heuristic, back-tracking search that applies one choice at a time and eliminates large subspaces of potential solutions when it can deduce a forbidden property, as specified by the constraints, in these solutions [34].

For the application of chromatic maze generation, the facts correspond to which color is assigned to each grid cell, and which cells are the start and end. The choice rules ensure exactly one fact is produced per grid position. This simple formulation can already create mazes, albeit ones where the end is possibly not reachable from the start. Desirable solutions can be assured by adding a constraint on reachability. One of the generated mazes is depicted in Fig. 10.

Compared to Ashlock's method, where the fitness function is essentially a black box, the ASP approach for maze generation is a "white box model" [7]. While the space of possible outputs is smaller than for a black box model, a designer can control it directly through constraints. For example, while the longest shortest path length is used to evaluate fitness in Ashlock's generator, the desired path length can be declared as a constraint in the ASP generator. Additionally, constraints provide a way to improve the quality of generated mazes, e.g., a designer can add constraints on which colors are traversed on all shortest paths [7].

Williams-King *et al.* [35] used a genetic algorithm (GA) combined with agent simulation to generate levels for a variety of *Loderunner*. The game can be considered a maze because the puzzler must discover a path through the level to collect all pieces of gold without encountering an enemy. The presence of moving enemies makes this maze dynamic, and the ability of the puzzler to dig indicates a path-building element.

This generation process has two phases that each employ a method of evaluation. The fitness function that steers the GA comprises direct static analysis, which alone cannot guarantee solvability but does allow for consideration of other interesting properties. Levels with high fitness are passed forward to the next phase where they are simulated with an AI player, to test the dynamic aspect, using 20 different paths. Difficulty is measured by the number of valid solution paths found in this process.

Williams-King *et al.* [35] state that their system can generate interesting levels in an online scenario. This two-part technique has potential to be applied to other puzzle games with dynamic elements. The scope for future work is mainly in the improvement of the evaluation function—an observation that can be made for several types of puzzles.

F. Path-Building Puzzles

This type of puzzle requires the player to build a path from a point A to a point B using a number of provided items. The path could be built for an entity in the world to traverse, such as an enemy AI; tubes, such as the *BioShock* minigame, shown in Fig. 11; a laser, such as in some *Portal* (2007) levels; or for the player character themself. This survey discusses gameplay paths, such as those undertaken by a player traversing a dungeon, in the narrative puzzle category, though one can also construe this as path building.

Path building is somewhat related to mazes but is differentiated by the nature of the game environment; in a maze, the player cannot change the environment, only find the best path through it, whereas in a path-building puzzle, the objective is to create a new path by altering the environment at the hand of tools or items. For some of these puzzles, the challenge stems from figuring out the correct placement of a limited number of items, whereas in others, there are more items than needed, i.e., decoy

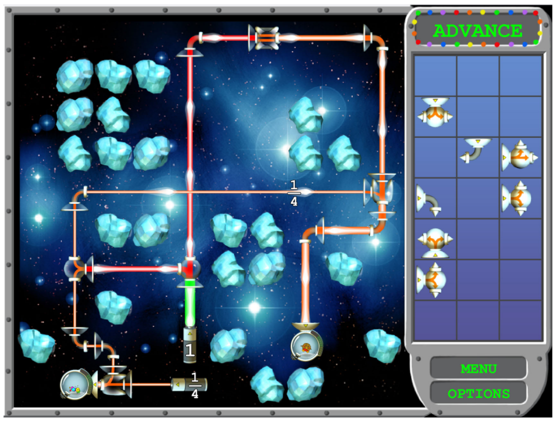


Fig. 12. Screenshot from *Refraction*.

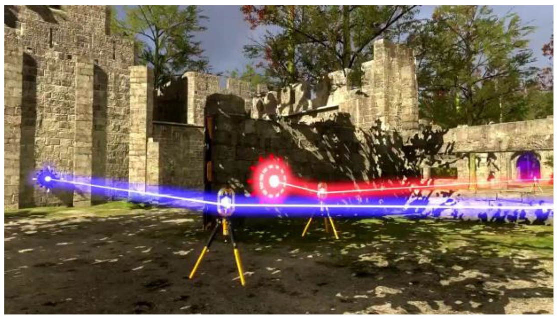


Fig. 13. Screenshot from *The Talos Principle* (2014), showing path-building with lasers.

items, and the player must determine which ones are most suitable, in addition to placing them. The latter usually lends itself to multiple possible solutions.

Smith *et al.* [10] studied the problem of hard constraints in procedural generation, e.g., that a generated puzzle is necessarily solvable, in the context of the path-building game *Refraction*. Many generators have not incorporated a way to guarantee that certain constraints, including aesthetic or difficulty concerns, are satisfied in their output. In other words, the constraints require the puzzle not only to be solvable, but solvable under certain prescribed conditions.

Refraction, shown in Fig. 12, is an educational game that aims to teach proportional reasoning (fractions) through a puzzle about redirecting laser beams toward a spaceships. The beams can be bent, split, etc., using different components, the placement of which also trains spatial problem-solving abilities. The game is set in a constrained space formed by walls of asteroids.

Smith *et al.* [10] present implementations of three constraint-based generation tools—mission generation, grid embedding, and puzzle solving—that each solve part of the problem of procedural puzzle generation. The first is responsible for creating a general outline with possible level solutions; it transforms a set of math expressions and a target number of pieces into a directed acyclic graph (DAG). Grid embedding translates a mission, specifically the DAG, to a geometric layout. Finally, puzzle solving aims to construct alternative reference solutions.

The overall level generator uses the structure defined by Dorman and Bakkes, which distinguishes between missions and spaces; missions are the logical order of goals the player must accomplish, whereas spaces are the physical layouts of the levels [36]. This distinction is valid for many puzzle games, and may be especially useful for approaching puzzles set in 3-D space, such as those in *The Talos Principle*. Part of a path-building from *The Talos Principle* is shown in Fig. 13; the mechanic of redirecting lasers to solve the puzzle is similar to that found in some *Portal* levels. While *Refraction* is a 2-D grid-based video game, the high-level formulation of the different components could lend themselves to a 3-D adaptation—mission generation would not need to change.

Smith *et al.* [10] made two implementations of each of the three tools. The initial implementation of mission generation uses a constructive feedforward approach where pieces are inserted one by one, and the generated mission DAGs all represent feasible solutions by construction. The first grid-embedding and puzzle solving implementations both used DFS. The main bottleneck in a system comprised of these tools was the grid-embedding step, which the authors identified as a constrained search problem that could not be fruitfully addressed with a generate-and-test approach.

The second implementation of the tools is based on ASP, and Smith *et al.* [10] found that these performed better than the original tools, both in terms of speed and output quality. The quality is better because the ASP approach allows for the specification of style constraints. They also found that declarative languages can be a powerful expressive tool for reliable, controlled puzzle generators. Constraint-focused generator design can allow aesthetic failures to be treated the same as absence of solvability. Few other generators reviewed in this paper have reached the stage of taking aesthetics into account.

In a later study, motivated by the sequel *Refraction 2*, Smith *et al.* [34] looked at constraining undesirable solutions, which they also refer to as shortcut solutions. In path-building puzzles, the unforseen combination of pieces, either those from the primary solution or those added as distractions, can introduce alternative solutions that were not intended by the designers. Since *Refraction* is an educational game, the designers require tight control over the puzzle progression, though this tends to also be a concern for game designers in general. Easier and/or alternative solutions, which could occur in *Refraction* as mathematical or spatial shortcuts, are potentially unacceptable, if they demonstrate a pattern that the designer did not anticipate.

Smith *et al.* formalize the generation of solvable puzzles with no undesirable solutions as an NP^{NP} complete search problem. To address this problem, they added two design-oriented extensions to their previous ASP approach [10] that allows the space of desirable, i.e., shortcut-free, solutions to be declaratively posed. This improved generation technique resulted in a reliable qualitative leap in the produced puzzles.

Butler *et al.* [37] used the ASP-based puzzle generator by Smith *et al.* as part of their tool for designing level progressions through mixed-authorship. The tool uses constraints, which are specified by designers and are incorporated into the original

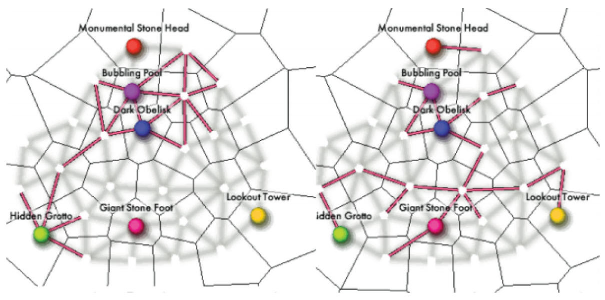


Fig. 14. Representation of *Anza Island*, showing two possible solutions for the constraints "Dark Obelisk cannot be connected to Monumental Stone Head. Monumental Stone head cannot be connected to Hidden Grotto. Bubbling Pool must be connected to Hidden Grotto" [9].

generator, to address the generation of a logical progression of levels. Combining a high-level view with the creation of individual puzzles is interesting, e.g., the incremental introduction of concepts.

Encouraged by Smith and Mateas' work on using ASP for the generation of mazes, as covered in Section II-E, Compton *et al.* [9] used ASP in making an adaptive procedurally generated puzzle game called *Anza Island*. In this game, a human plays against an AI, Anza. The human must visit a number of landmarks in different zones to win, while Anza attempts to thwart this by controlling, i.e., activating and deactivating the bridges between zones. The human player must create a number of constraints, based on collected logic cards, that force Anza to create the bridges they require (this is the path-building element). Two possible solutions for three given constraints are shown in Fig. 14. This novel approach to PCG directly incorporates the generation into the game—the constraints specified by the player modify the answer set program at runtime. The declarative nature of ASP makes it possible to ensure all generated content would be valid.

G. Narrative Puzzles

Narrative puzzles can be defined as puzzles that form part of the progression of a narrative, whose solutions involve exploration and logical as well as creative thinking. They are a key component of adventure and story-driven games, and often feature in large open world games, including RPGs. Narrative puzzles can be viewed as temporary obstacles to the narrative's advancement, though they do not always have to be solved in a precise order.

Adventure games have many subgenres, including text adventures, point and click adventures, and escape the room games, such as *The Room* (2012) series. Not all of the puzzles in this range belong in the narrative puzzle class; adventure games could choose to include any of the puzzle types mentioned in this classification. However, narrative puzzles is one of the broader puzzle classes; it incorporates many logical thinking and pattern matching type puzzles, such as those in *Myst* (1993).

A good narrative puzzle should have a satisfying solution, i.e., one that ultimately makes sense to the puzzler. Puzzlers find solutions by exploring the environment and investigating ways in which objects can be manipulated and combined. Some examples of narrative puzzle patterns, as outlined in [38], are as follows.

1) Figuring out which item a character desires, usually leading to a reward in exchange.
2) Logically combining two objects to change their properties or to create a new object.
3) Disassembling an object into useful components.
4) Saying the right thing to convince a character to provide aid.
5) Acquiring a key to open a new area.

Unexplored (2017) is a published procedurally generated roguelike that integrates varied, generated lock and key- and-code based puzzles [39].

The use of the label "narrative puzzle" could be debated because it is possible to switch out the specific puzzles without affecting the overall narrative—the focus of the research in [40]. However, the label has been used in past research and fits when viewing story progression as centered on player action.

A different take on a narrative puzzle is *Framed* (2014), a puzzle game set up like a noir comic in which the player must rearrange the order of events to prevent the protagonist from getting caught or shot by the police. As some of the rearrangements are focused on creating physically feasible path, rather than a narrative one, this game is considered a blend of a narrative and path-building puzzle.

The puzzle-dice system was developed by the MIT Media Lab Singapore for the purpose of generating narrative puzzles [38]. Their motivation was to add replayability to adventure games, which frequently have only one linear path. The use and development of the puzzle-dice system is demonstrated through two proof-of-concept games: *Symon* (2010) and *Stranded in Singapore* (2011).

Symon was an early prototype with a dreamlike setting that allowed for fantastical logic, i.e., using something cold to change an object's color to blue. It was small in scope but succeeded at demonstrating the generation of a short, replayable game. The initial system for *Symon* used a puzzle map that concatenates narrative puzzle patterns into the structure of a game. Both this map and the possible puzzle patterns need to be specified by a designer; the generation process then inserts characters and objects into patterns, and patterns into the map, using a brute force approach. This grammar-based method of generation ensures that all puzzles will be solvable because the generation is constructive.

The full puzzle-dice system is more flexible than the initial one built for *Symon* and gives designers more control. It was used to create *Stranded in Singapore*, shown in Fig. 15; an adventure game in which the puzzler must fulfill the demands of a number of characters, often through logical combination of items found in the world. The full system is based on puzzle building blocks that can be customized by designers; a modular approach that aims to allow for expansions. This system's puzzle map is a map structure resulting from the combination of building blocks and defines the chronology of actions the players must undertake, in terms of dependencies.

Puzzle maps are paired with a database of designer-defined game items that have attributes and relationships to other items.

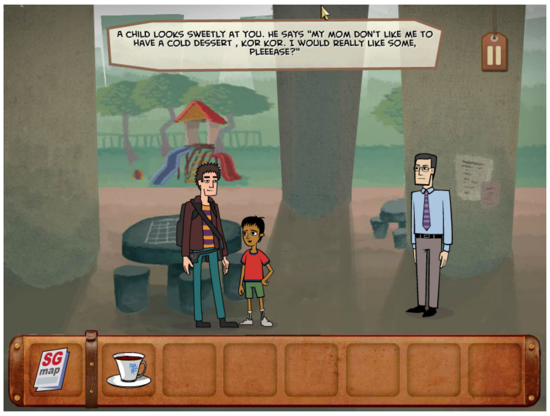

A CHILD LOOKS SWEETLY AT YOU. HE SAYS "MY MOM DON'T LIKE ME TO HAVE A COLD DESSERT , KOR KOR. I WOULD REALLY LIKE SOME, PLEEEASE?"

Fig. 15. Screenshot from *Stranded in Singapore* [38].

The relationships are used to determine which items can fit into certain puzzle building blocks. For example, each item has a "madeby" property that links it to two other items that can be combined to create that item. Building blocks perform generation of narrative puzzles independently given a desired output. The generation process is composed of three steps: generating the output item with the desired properties, generating inputs, and finally creating a relationship between the input and outputs. The possible relationships include: combine, property change, insertion, request, and area connection.

The motivation behind the development of the puzzle-dice system [38] is threefold; the researchers wanted to create a tool that guarantees solvability in its output, is accessible to designers, and is general enough to allow for a range of narrative-type puzzles. The tool is more like a framework than an out-of-the-box puzzle generator as both the item database and puzzle map must be constructed by the designer. Narrative puzzles, by nature, require a large amount of designer input compared to other puzzle classes.

Dart and Nelson worked on adventure-game puzzle generation using smart terrain causality chains (STCCs) [40]. They focused on creating a drop-in solution for the issue of replayability: generating variations on puzzles that fit in the same place in an existing story line. In other words, this technique introduces replayability without high-level narrative variation, and stands in contrast to the common branching story lines approach. The puzzles do change the low-level narrative significantly and, thus, fit this puzzle class.

The puzzle generation relies on a database of smart terrain items to create causality chains that form puzzle solutions. Smart terrain items, introduced by Will Wright, developer of *The Sims* (2000), have a set of associated actions and properties that determine how they behave and how the environment can affect them. In contrast to the puzzle-dice system, items are not aware of their specific relationships to other items. Interactions between objects are executed through use of physics simulations, such as phase transitions, collisions, or indirect energy [40].

An STCC is a directed graph that defines dependencies between all the objects in the scene and the objective of the puzzle [40]. The sequence of items represented by the STCC corresponds to the sequences of actions that the puzzler must perform to solve a puzzle. A list of possible actions for an item, along with the causes and effects, allows the creation of an STCC using a backward chaining algorithm. The generation starts from a set of scene objectives and runs till it reaches primitive smart-terrain objects, which are then placed in the scene. This is a constructive process that guarantees at least one solution, and that each item in the scene is relevant for at least one solution.

Dart and Nelson tested their generation method in the adventure game *Space Dust* (2014). Players have to replay this game several times to acquire all the information needed to win, and on each playthrough, have to solve different puzzles to progress. They discovered that 70% of players found the puzzle scenarios easier when there were more possible, parallel solutions, and most players said that longer causality chains corresponded to more difficult puzzles [40]. Their experiments also showed that players found the game engaging, despite the repetition in overall story.

Object placement in the physical game world is currently not automated, but could be added in by also storing semantic information about the environment. Another area of future work is actions/effects inference; similar to in the puzzle-dice system, all cause and effect information has to be manually specified by a designer, but it would be desirable to automate this process.

Generally, procedural generation for dungeons has focused on the level layout, but van der Linden *et al.* [31] also looked at generating sequences of player actions for a gamed called *Dwarf Quest* (2013). These actions include fighting enemies as well as basic narrative puzzle constructs, such as key-and-lock challenges.

An action graph is generated from a gameplay grammar in which player actions are defined as a verb and a target, which is typically a game item. Additionally, actions may have sequences of subactions, which may represent disjoint alternatives. For example, a subaction of unlocking a door with a key might be fighting an enemy from whom the key can be looted. Designers can control generation through the authoring of the grammar, and through the initial actions, since those steer the recursive substitution of compound actions with subactions (or subgraphs). The produced action graphs are converted to physical dungeon levels, but that aspect falls outside the scope of this survey.

Another category of PCG that can fall under narrative puzzles is quest generation. Quests are commonly used in RPGs as plot advancement devices but tend to represent higher level narrative elements than the narrative puzzles previously discussed in this section.

Doran and Parberry analyzed the structure of almost 30 000 quests from four popular MMORPGs—*Eve Online* (2003), *World of Warcraft* (2004), *EverQuest* (1999), and *Vanguard: Saga of Heroes* (2007)—to develop a quest generator [41]. They determined that the structure of a quest, which a player obtains from an NPC, can be described by: the NPC's motivation, the strategy, as assigned by the NPC, for completing the quest, and the individual sequence actions that lead to the implementation of these strategies. From the surveyed games, the top NPC motivations appear to be conquest, equipment, protection, and knowledge. Some example strategies are "spy," "attack enemies," and "treat or repair," and some possible actions are "go to" and "give."

The researchers use a context-free grammar in Backus–Naur form to describe the quest structure formally. The terminals of this grammar correspond to the atomic actions that can be undertaken by a player. As such, quests can be expressed as trees, where the root represents the entire quest, and the leaf nodes, visited in preorder traversal of the tree, detail the sequence of player actions necessary to complete the quest.

The quest generator starts by randomly picking a motivation and selecting an associated strategy. This becomes the root of the tree, and from there nodes are recursively replaced according to the grammar rules to generate substrategies (subquests) and actions. The choice of rule to use for a replacement is dependent on the state of the game at that point to, for example, prevent creating a quest for obtaining an item that player already has.

Many RPGs use recurring quest structures in which the details are just replaced to generate "different" quests; Doran and Parberry propose that their generator can also change the structure, to provide a wider range of useful content. However, in the cited paper, they had not evaluated whether the quests created by their generator were equal in quality to hand-crafted quests [41].

H. Physics Puzzles

Puzzlers have to use game physics to complete the puzzles belonging to this class. Physics in games is (most often) modeled after real-life physics, and so finding a solution requires players to predict the physical outcomes of possible actions. These puzzles naturally only exist in digital games. Physics puzzles have an element of unpredictability, as the game environment changes in real-time according to the laws of physics.

In contrast to many other puzzle classes, the solutions to these puzzles may require precise timing and/or execution of actions on the player's part; for example, shooting a new portal while falling in a *Portal* puzzle room. This is dexterity difficulty, modeled by Isaksen *et al.* [42] as one of two orthogonal components of perceived difficulty—the other being strategy. Dexterity difficulty is an element of interactive puzzle games and speaks to the execution of a solution. In comparison, strategic difficulty refers to the derivation of a solution and is influenced by the size of the search space and the required application of logic. The dexterity component poses a challenge for generating physics puzzles because a generator must ensure solutions are feasible, i.e., they have a margin of error that accommodates human reaction time. Diversely, viewing the dimensions independently may allow a generator to create puzzles that are tuned for a certain skill.

Shaker *et al.* [43] focused on the generation of levels for physics-based puzzle games, using a clone of the mobile game *Cut The Rope* (2010) as a test ground. The goal in *Cut The Rope*, shown in Fig. 16, is to make the candy drop in such a way that it reaches a frog monster placed at a fixed position. There are different game objects that can be used to change the movement direction of the candy, including ropes, air cushions, bubbles,

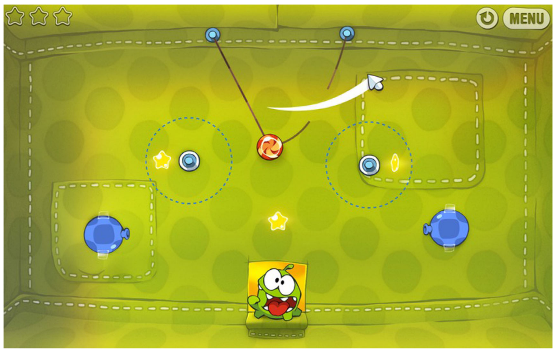


Fig. 16. Screenshot from *Cut The Rope* [43].

bumpers, and rockets. All these objects obey the laws of Newtonian physics.

The game generator by Shaker *et al.* [43] evolves levels based on a context-free grammar, which is a set of recursive rewriting rules. Design grammars offer a concise way of describing a huge variety of possible level structures. Grammatical evolution (GE) is a technique that combines an evolutionary algorithm with a grammatical representation. They constructed the grammar by analyzing original levels for design patterns.

Levels, which are the phenotypes, are represented by lists of objects that can be placed anywhere in the map and may have some properties. As mentioned, the structure of a level is described by a design grammar, which is what is used to evolve levels. The GE process includes a genotype-to-phenotype mapping using production rules from the design grammar. This results in phenotypic programs that are syntactically correct—these programs are then evaluated for fitness.

A fitness function based on direct heuristic measures, which are theory driven, is used to rate and consequently evolve levels. However, high fitness is only an indication of playability, not a guarantee. Consequently, Shaker *et al.* experimented with a simulation-based fitness function that works by randomly selecting actions based on game state until the game is won, lost, or times out.

To estimate the expressiveness of their generation, which corresponds to the range of all levels, the researchers defined three measures: frequency, orientation, and density of game items. They saw promising results in terms of efficient exploration of the content space, and also reflections of the constraints imposed by their grammar and fitness functions.

In later research, Shaker *et al.* [44] developed a deliberative Prolog-based agent to improve the simulation-based evaluation component of their original generator. The agent uses inference to determine a set of sensible actions and a depth-first search is then performed on this (reduced) search space. Two different rule sets were used for the agents; the first focused on the all game objects' properties and their placement in the level, whereas the second contains only objects the candy can reach while in a given position, direction, and velocity. The use of the notion of reachable components in the second ruleset greatly improved the processing speed, because it disregarded more branches. Overall the evaluation technique is effective at detecting playable levels, but it does not address difficulty.

Finally, Shaker *et al.* [45] researched a progressive approach for content generation; this approach combines constructive and search-based approaches to create a system that is fast, flexible, and reliable. The progressive approach has two components: timeline generation, where GE is used to evolve a timeline of in-game events, and timeline simulation, where the timelines are simulated using an intelligent agent. The agent maps the timeline to a possible level layout in a constructive manner and assigns it a fitness based on level design, playability, and aesthetic.

The progressive approach provides a one-to-many mapping between genotypes and phenotypes; each timeline (genotype), which described the rhythmic feel of the game, could be mapped to several different level layouts (phenotypes). The timeline simulation component combines the generation of the complete design with a playability check, and guarantees that all placed components are needed in the completion of the puzzle.

Compared to their previous approaches, Shaker *et al.* found that the progressive approach was much faster and resulted in a greater variety of generated puzzles. They state that this technique could be applied to many games to induce a significant decrease in generation time. Future work includes providing a way to control difficulty as part of the timeline generation.

Ferreira and Toleda used a genetic algorithm to generate levels for another popular 2-D physics puzzle game: *Angry Birds* (2009) [46]. Briefly, the objective in *Angry Birds* is to crush all the pigs, in the allotted time, by using a slingshot to throw birds against a structure of blocks and pigs. The player must use physics knowledge to estimate how hard and at what angle to fling the bird to trigger an optimal collapse of the structure. Since the source code of *Angry Birds* is not available, the researchers created a clone using the *Unity* game engine.

The game clone is used as part of their genetic algorithm to perform simulation-based evaluation. The structure of candidate levels is encoded as an array of columns with elements (block, pig, or composed block), but at the evaluation stage, all the individuals in the current population are transcribed to an XML file that can be used to build the level for simulation. The results of each simulation—specifically the average velocity and number of collisions for the blocks and pigs, plus end rotation for the blocks—are recorded for use in determining the stability of structures in a level. The genetic algorithm considers the most fit levels to be those that are the most stable and performs crossover and mutation by manipulating the associated arrays of columns [46].

An important parameter in these simulations, which the researchers determined experimentally, is the time limit: if the simulation time is too short, game items may not have started to fall when the timer runs out; if it is too long, all elements will end up at rest. The algorithm initializes the population according to some probabilities that prevent randomly creating start individuals with extremely unstable structures.

Ferreira and Toleda, such as Shaker *et al.*, defined metrics for evaluating the expressivity of their level generator, specifically frequency, linearity, and density, which refer to the blocks' types and orientations. These metrics showed that there was variation in the generated levels, e.g., structures with varying heights.

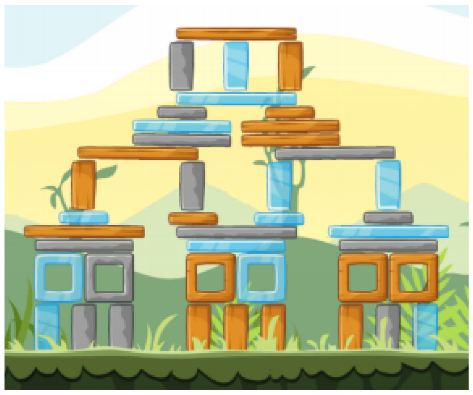


Fig. 17. Example of a generated structure for an *Angry Birds* level, from [47].

Stephenson and Renz came up with an algorithm for generating *Angry Birds* levels that is dependent on a self-contained structure generator [47]. The structure generator creates independent structures, using a variety of different blocks, for different randomly sized subsections of the level. This generator works top down: it starts by picking a block for the peak of a structure, and then recursively determines blocks to place underneath. Block choices are made at the hand of a probability table and a validity check, for the stability of the structure, is used to determine the layout of the blocks. An example of a fully generated structure is shown in Fig. 17.

After the structures have been placed, they are analyzed to determine possible pig placement locations. This is followed by a structural analysis to determine weak points that should be protected from the player, by the introduction of extra blocks, so that the level is not too easy. Finally, a number of birds is added, which also affects difficulty in that it determines the number of "shots" the player gets.

This generator is highly controllable by a designer as there are many inputs, such as the block selection probabilities and number of structures that can be customized. Stephenson and Renz also used a number of metrics to evaluate their generator: frequency of block types, structural height variety, density, difficulty, which is dependent on the input parameters, and playability. The researchers concluded that their generator is flexible enough to be applied to other games, and that it can come up with a broad range of levels [47].

I. Logic Puzzles

Logic puzzles are solved through deductive reasoning; the player arrives at a solution through a series of deductions made from some given premises. The first logic puzzles appeared in Carroll's book *The Game Of Logic* [48] and were akin to syllogisms. Popular puzzles in this category are *Sudoku*, in which the player makes deductions about placements of numbers, and logic grid puzzles. The latter is so named because the puzzler is provided with a grid on which to fill in information obtained from clues.

There is a significant amount of research on, and commercial interest in, the generation of *Sudoku* puzzles, likely due to their popularity in newspapers and magazines. Some of the research is summarized ahead.

Mantere and Koljonen used a genetic algorithm to generate, rate, and solve *Sudoku* puzzles [49]. As for some other approaches documented in this survey, this method closely relates puzzle solving and generation. However, Mantere and Koljonen use evolution to solve the puzzle instances, whereas it has been more commonly seen as part of puzzle generation. The genotype used for evolution is an array of 81 integers, in blocks of nine. Crossover can only occur between blocks, whereas mutations can only occur within blocks. A puzzle is created by randomly drawing 20–50 fixed numbers from a Sudoku solution and then solving that instance up to 10 times to check that only one solution exists. While GA is not the most efficient way to solve a *Sudoku* puzzle, experiments showed that the number of generations required in solving is a good indication of difficulty [49].

Boothby *et al.* [50] generated *Sudoku* puzzles by applying the inverse of solving methods; they started with a completed *Sudoku* board and worked backward. A constrained BFS search, with heuristics to prune the search space, was used to ensure that each puzzle would be uniquely solvable. Puzzle difficulty is rated according to the solving techniques used in generation, based on the fact that puzzlers will apply simpler techniques first. Notably, this technique allows for generating puzzles of desired difficulty, an uncommon characteristic in puzzle generators.

A similar method was applied by Xue *et al.* [51]. They constructed a valid board using the Las Vegas algorithm and then applied a "dig-hole" strategy that effectively corresponds to a pattern-based erasure of some of the digits on the board. Gebser developed a little-known Sudoku generator that uses ASP to create puzzles that are guaranteed to have a unique solution. Additionally, all clues in the generated puzzle are essential, meaning that the player would reach an alternative solution if a clue was removed. Generally, this also implies a Sudoku level with a low number of clues and, thus, a relatively high level of difficulty. Gebser's program uses choice rules, disjunction, and cardinality constraints to specify a generator in less than 40 lines of code [52].

Finally, Fatemi *et al.* [53] generate *Sudoku* puzzles of different levels of difficulty using a hill climbing approach. The evaluation of board difficulty is based on constraint satisfaction problems (CSPs). A CSP can be defined as a set of variables, all the possible values for those variables, and the constraints between one or more of the variables. The solution to a CSP is an assignment of values to variables that respects all the given constraints. They use consistency techniques with domain splitting, meaning the search space is reduced through removal of values that cannot be in the solution, and the problem is decomposed into smaller problems. The number of times a consistency call is made correlates to the difficulty of the puzzle.

Shinro, shown in Fig. 18, is a Japanese logic puzzle in an 8 × 8 grid similar to *Sudoku*. To solve it, players must determine the locations of 12 stones using two types of clues; the number of stones located in each row and column, and arrows placed in the grid that point to at least one stone. As is common for logic puzzles, the player uses elimination to reach one of two conclusions; a square must contain a stone, or it absolutely cannot contain a stone.

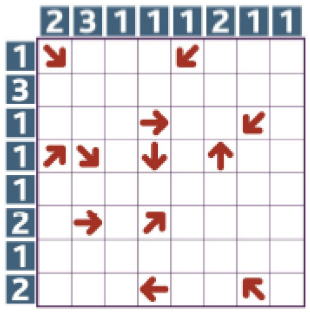


Fig. 18. Example *Shinro* puzzle.

Oranchak used a genetic algorithm to automatically generate *Shinro* puzzles with desired qualities [54]. The basis of the evolutionary algorithm is an automated solver that uses a list of possible deductions to solve *Shinro* puzzles. Among others, deductions are made based on the following observations: row or column count is satisfied, arrow points to only one free square, and locations that can be excluded based on the pigeonhole principle. Some deductions are easier to make than others, so it is reasonable to say that the difficulty of a *Shinro* puzzle, and a *Sudoku* puzzle, can be measured by the difficulty of the required deductions.

The automated solver tries to solve the generated puzzles using a greedy search, i.e., one that looks for easy moves first. During the search, the number of times each type of move is executed is tracked, and these numbers are then used to compute fitness.

The evolutionary algorithm is initialized with a small population of genomes because the fitness evaluation has a long run-time [54]. Genomes are matrices of integer values representing different grid square types. It is also possible to specify a target pattern that will restrict the possible values for some grid squares, offering designers a way to influence the output of the generator. Evolution occurs through mutation, which probabilistically changes the values of randomly selected squares and randomly decides whether to employ symmetry. When the algorithm stops due to no more improvements in fitness, a brute force search is used to make sure the resulting puzzle only has one unique solution.

Oranchak found that optimizing puzzles according to the number of required moves did not necessarily lead to difficult puzzles. As mentioned, the more important factor is how challenging the moves (deductions) themselves are, so concentrating on maximizing specific types of moves made more sense. However, this is challenging due to the nature of the greedy search. Aesthetics, such as symmetry, can make a puzzle more appealing to a player, and introducing the symmetry mutator greatly contributed to generating puzzle with symmetry in the stone and arrows positions.

J. Word Puzzles

This is a broad category that incorporates any puzzle that is dependent on words. Some puzzles only exist as word-based puzzles, e.g., riddles, crossword puzzles, and anagrams, but many word puzzle formats also have numerical or symbolic equivalents, e.g., analogies and odd-one-out puzzles. Different subcategories of word puzzles that are based on the semantic similarities, or dissimilarities, of words pairs have a lot in common. For example, odd-one-out and next-in-sequence word puzzles require the player to discover the most plausible concept that relates a set of words [55].

Word puzzles have a long history on paper, but digital adaptations and novel-types of these puzzles are now a popular category of mobile games. Procedural generation is useful for word puzzles due to specific word-related issues: new words get created, old words go out of common use, and existing words get new meaning [55].

Colton investigated three types of word puzzles: odd-one-out, analogy, and next-in-sequence [4]. These puzzles have a common characterization: each puzzle consists of a question, a set of choices including the answer, and an embedded concept, i.e., a single plausible solution. Finding a solution is generally based on player's previous knowledge about the characteristics of the objects, but obtained by systematically exploring the solution space.

Colton proposes that the difficulty of these puzzles can be influenced by the number of choices, the complexity of the solution concept, and the number of disguising concepts, i.e., concepts that may look like a possible solution at first glance. He extended a system for automated theory formation in mathematics [56] to generate puzzles [4]. This system takes one or two existing concepts as input and uses production rules to invent a new concept.

The generating process is largely similar for the three different kinds of puzzles and while generally constructive, a check is required after generating each puzzle to ensure that the solution is the simplest one that exists. The main challenge was ensuring each generated puzzle had only one solution, which needs to happen reliable in order to use complex embedded concepts. Colton's approach relies on highly structured datasets, which requires a lot of human annotation effort in comparison to the system by Pinter *et al.* [55].

The automated word puzzle generator developed by Pinter *et al.* [55] uses: a corpus, an unstructured, and unannotated document collection; a topic model, which creates a topic dictionary from the input corpus; and a semantic similarity measure of word pairs. The semantic similarity is computed using a concept repository; one such repository used by the authors is Wikipedia. A word is considered to be closely related to a certain concept if it appears frequently in an article about that concept. This method would allow for domain specific puzzles, e.g., to fit into a narrative, by using a relevant corpus. Odd-one-out puzzles are produced through two consecutive algorithms. The first identifies consistent sets of words by generating candidate consistent sets and keeping or discarding them based on the similarity of the two most dissimilar words. The second algorithm combines each set with a weakly related, or unrelated, word. The quality of the word puzzles is closely linked to their difficulty, and future work includes testing whether humans find the generated puzzles interesting.

While the described generation methods output word puzzles, the words could be replaced by objects to place the puzzle in a 3-D environment. This would only impose restrictions on the types of words that can be used as the choices.

Crosswords are a popular topic for puzzle PCG for the same reasons as *Sudoku*; both are still prevalent in print media. Rigutini *et al.* [57] present a system that can generate crossword puzzles without human intervention. This system extracts definitions from the Internet using natural language processing techniques and compiles a valid crossword schema using CSP solving in a similar fashion as discussed for logic puzzles in Section II-I.

K. Riddles

Riddles are puzzles that are solved by finding a plausible explanation for an unusual description, often posed as a question, and require the puzzler to use lateral thinking. The solution, or answer, is rarely immediately obvious, but should make sense to the puzzler upon its discovery. Riddles feature in *The Hobbit* by J. R. R. Tolkien, e.g., "A box without hinges, key or lid, Yet golden treasure inside is hid," to which the answer is, "An Egg" [58]. It is uncommon to see riddles in video games because parsing a natural language explanation is difficult for a computer. Exceptions are cases in which the answer to a riddle does not need to be stated but rather gives players a hint as to an object they need or a direction they should travel in.

Riddles are difficult to generate because they often rely on a play of words—a difficult concept for computers. Galvan *et al.* [59] generated riddles of the format "What is as hot as soup?" and stringing several such comparisons together to make the riddle solvable. Only one comparison leads to too many possible answers, and would be impossible to solve without turning into a lengthy guessing game. The most well-designed riddles are based on nuanced meanings of words, and the generator in [59] does not possess that level of semantic power.

Galvan *et al.* [59] used the Thesaurus Rex, a database of word associations that assigns words categories and attributes, according to their use in everyday language. Each category and attribute has an associated weight for each concept (word). The riddle generation process involves finding links between the target concept and other concepts using the categories and attributes. Difficulty of the riddles can be adjusted by changing the threshold of how similar concepts must be to be used as one of the comparisons.

The randomly generated riddles sometimes suffered from comparisons that did not add new information, and new comparisons that were contradictory to old comparisons due to the polysemy of some concepts. These issues feed into the problem of ambiguity and having one best plausible solution (a satisfying answer), a challenge that also exists for other categories of word puzzles.

Guerrero *et al.* [60] developed a Twitter bot that generates riddles about celebrities and well-known characters by extracting information from sources such as Wikipedia. The riddles are composed of vague descriptions of the attributes of the person or character in question. Sometimes the riddles do include nonliteral meanings of words, but overall their standard is not high, and often too complex, as only about 16% if users found the correct answer. It is difficult for a generation program to achieve a satisfying level of nuance.

III. Analysis

In the framework of a puzzle classification, we have examined 32 projects on procedural generation of puzzles in detail. These projects represent a variety of approaches and puzzle types. In an effort to provide a complete picture of this research field, a summary of the observations made about the different puzzle generation methods—in terms of the characteristics for PCG, outlined in Section I-B—is given in Table I. The table covers most of the approaches mentioned in the survey, but due to space constraints, leaves out a few that were only briefly mentioned.

We can make some observations across the puzzle types for each characteristic of PCG methods, i.e., for each column. Looking at the first column, search-based and generate-and-test approaches, as described in Section I-B1, appear slightly more popular overall. For some categories there is a distinct emphasis on a single technique being employed—for example, all the narrative puzzle generators we looked at use a constructive approach. We also noted that combining techniques has been met with some success.

From the evaluation column, we can observe that simulation-based testing is used in 7 of the 11 generate-and-test methods, possibly because puzzle solvers, which can be used for testing, are also a major area of research. Simulation-based testing is particularly common for physics puzzles, likely due to the dexterity component that is not present for most other puzzle types. An evaluation step is not usually applicable to constructive methods, as was mentioned in Section I-B2. Constraint-based techniques have been labeled as such to distinguish them from independent types of evaluation.

Most current approaches are intended for offline rather than online generation. The offline trend could be a reflection of the fact that many works are experimental and are focused initially on developing methods for generating interesting content of reliable quality.

Generally, puzzle types that tend to have simpler rules, e.g., *Sokoban*-type puzzles, have less designer input requirements than an open-ended category, such as narrative puzzles. Generation methods that require designer input are more like frameworks than out-of-the-box generators. Narrative puzzles require the most designer input as they are loosely defined, and dependent on creativity and common-sense logic—challenging domains for PCG. Most methods that provide a medium degree of control do so through templates or fitness function parameters.

While mixed authorship is popular for PCG in general (e.g., for dungeon generators), it is only used by 5 of the 32 puzzle generation methods. However, generators with a high degree of control can additionally be viewed as a collaboration between designer and algorithm. Some papers make no reference to mixed authorship, and in this case, we assume automatic because many puzzles (e.g., *Sokoban*) would potentially become unsolvable even by minor designer tweaks to the generated content.

There are very few approaches that allow puzzles to be generated in an adaptive fashion, i.e., where the player's actions are used as input to the generator. There is clear scope for future work that attempts to generate puzzles tailored to individual players.

The first step for any puzzle generator is to ensure solvability of the generated puzzle. In the quality consideration column, we list notable quality-focused features that go beyond solvability. Most generators attempt to estimate quality after generation has completed, but only constraint-based generators reliably generate puzzles with given, desired qualities.

In addition to the immediate observations we have made from Table I, the intention is also that this review will serve as a map of the state of the art in the procedural generation of puzzles. To facilitate this, we pinpoint "gaps in the map," i.e., promising areas for future research.

A question is why ASP has been applied in some places and not others. One requirement of ASP is of course that it must be possible to specify constraints (e.g., like in *Sokoban*), but this may not be possible for all puzzles. In terms of general applicability, Smith and Mateas observe that ASP is "applicable to problems where the task is to select structures with desirable properties from a vast but countably finite space" [7, p.193]. Additionally, it is plausible that in some cases, ASP has not been tried because it requires learning an unfamiliar language paradigm; software libraries may make constraint solving more accessible through familiar paradigms. Since puzzle rules can be considered constraints, constraint-based techniques are a promising future direction for puzzle PCG.

One "gap" is formed by the lack of application of techniques that can execute in an online environment. Decreasing the generation time while maintaining quality and solvability is an important challenge to overcome in the face of integration of procedurally generated puzzles into commercial games. This could be achieved by improving the performance and reliability of some of the presented techniques, or by trying alternative techniques that are known to work online. Puzzles can provide a more surprising and interesting experience for players when they are created on the fly, i.e., while the player is progressing through the game. Some puzzle types, such as narrative puzzles, could even be tailored according to the player's style of play and previous in-game choices.

In addition to the "gaps in the map" that are clear, there are of course also gaps that are not shown in the table, namely the puzzle types that were not covered in this survey, because they have only recently been subject to PCG, or not yet at all. This includes time manipulation puzzles, i.e., puzzles in which the puzzler must use time as a puzzle solving mechanic. For example, *The Talos Principle* includes puzzles that require the player to find a solution using parallel timelines. *Braid* is perhaps the most well-known game to use time manipulation as its core mechanic—each world of the game has a distinct time-based concept, such as linking time to the player's location. Procedural generation for time manipulation puzzles would be challenging

TABLE I
SUMMARY OF PUZZLE GENERATION METHODS

Puzzle Category	**Specific Method**	**Constructive vs Generate-and-Test**	**Evaluation (Direct vs Simulation-based)**	**Online vs Offline**	**Degree and Dimension of Control**	**Automatic Generation vs Mixed Authorship**	**Generic vs Adaptive**	**Quality Consideration**
Sokoban-Type Puzzles	Random template combination + BFS test [17]	Generate-and-test	Simulation-based: automatic solver	Offline	Level templates	Automatic	Generic	Uninteresting candidates removed
	Random template combineation + heuristic-guided search [14]	Search-based	Metrics from backwards play	Offline	Level templates	Automatic	Generic	Rejects easy/similar levels
	MCTS using gameplay moves [15]	Search-based, with simulated gameplay	Direct: level layout metrics	Online possible	Eval. function	Automatic	Generic	Quality improves incrementally
	Data-driven MCTS [19]	Search-based, with simulated gameplay	Direct: features correlated with difficulty	Online possible	Eval. function	Automatic	Generic	Addresses difficulty
	Large-scale BFS with endgame data [20]	Search-based	Direct: using endgame database	Offline	Not much control	Automatic	Generic	Addresses difficulty
Sliding Puzzles	Symbolic methods with binary decision diagrams [22], [23]	Search-based	N/A	Offline	Not much control	Automatic	Generic	Tries to find difficult levels
Tile-Matching Puzzles	Random placement + BFS validation [25]	Generate-and-test	Simulation-based: solver	Offline	Level editor	Mixed authorship: designer can modify the level, then re-test	Generic	Item placement aesthethics
Assembly Puzzles	Formal model for recursive interlocking [26]	Constructive	N/A	Offline	Any voxelized shape can be given as input	Automatic	Generic	Only one sequence of assem-bling/disassembling
Mazes	Evolutionary algorithm with dynamic programming [27]	Generate-and-test	Simulation-based using dynamic programming	Offline	Modification of fitness function	Automatic	Generic	Diversity promoting measures
	Genetic algorithm [35]	Search-based	Direct followed by simulation-based	Online possible	Fitness function could be tweaked	Automatic	Generic	Difficulty is measured
	Exhaustive search [28]	Generate all possibilities, then select/test	Direct	Offline	Not much control	Automatic	Generic	Tries to break learned heuristics
	ASP [7]	Search-based	Constraint-based	Offline	Designer can directly modify constraints	Automatic	Generic	Quality constraints
Path-Building Puzzles	Feed-Forward + DFS [10]	Constructive (pieces inserted one-by-one) then search-based	N/A	Offline	Not much	Automatic	Generic	Not much
	ASP [10]	Search-based	Constraint-based	Offline	Designer can modify constraints	Automatic	Generic	Style/quality constraints can be specified
	ASP with additional design-oriented extensions [34]	Search-based	Constraint-based	Offline	Designer can modify constraints	Used in mixed authorship tool for designing level progressions [35]	Generic	Can constrain undesirable solutions
	ASP [9]	Search-based	Constraint-based	Online	The player can modify constraints	Mixed if player is an author	Adaptive	Quality given by space of possible constraints player can specify
Narrative Puzzles	Grammatical structure + designer created database [38]	Constructive	N/A	Generation runs at start of game	High degree of control through rule and item database	Automatic	Generic	Quality is highly dependant on designer input
	Smart terrain causality chains [40]	Constructive	N/A	Generations runs at start of game	Possible actions are authored by designers	Automatic	Generic	Surveyed features players found difficult
	Gameplay grammar [31]	Constructive	N/A	Online possible	Authoring of the grammar	Automatic	Generic	Quality depends on grammar quality
	Context-free grammar [41]	Constructive	N/A	Online possible	Authoring of the grammar	Automatic	Generic	Quality has not been evaluated against hand-crafted quests
Physics Puzzles	Grammatical evolution + smart agent [43]	Generate-and-test	Two fitness functions: one direct, one simulation-based	Offline	Modification of fitness functions	Intended as an authoring tool for game designers	Generic	Fitness function can give weight to quality considerations
	Progresive approach [45]	Combination of constructive and search-based	Simulation-based fitness function	Offline	Modification of fitness function	One-to-many genotype-to-phenotype mapping is useful for designers	Generic	Fitness score takes design constraints into account
	Genetic algorithm [46]	Generate-and-test	Simulation-based	Offline	Fitness function be modfied	Automatic	Generic	Attempts to make challenging puzzles
	Indepdent structure generator + placement [47]	Structure generator is constructive, then the algorithm is generate-and-test	Direct: structural analysis	Offline	Many inputs can be tweaked	Automatic	Generic	Attempts to make challening puzzles
Logic Puzzles	Random placement + genetic algorithm based solver [49]	Generate-and-test	Simulation-based: solver	Offline	Fitness function used by solver could be tweaked	Automatic	Generic	Number of generations in solving indicates difficulty
	Backwards heuristic BFS [50]	Constructive search-based	Direct: heuristics	Offline	Variation of heuristics	Automatic	Generic	Can construct with desired difficulty
	ASP [52]	Search-based	N/A	Unclear	Stand-alone, constraints already specified	Automatic	Generic	Guaranteed unique solution, all clues essential
	Genetic algorithm + solver for fitness function [54]	Generate-and-test	Simulation-based: solver (fitness function uses direct evaluation)	Offline	Fitness function, optional target pattern constraint	Automatic	Generic	Incorporates aesthetic considerations, some control over difficulty
Word Puzzles	Automated theory formation using production rules [4]	Constructive + check	Direct	Offline	Highly structured data sets	Automatic	Generic	Ensures a single solution + ids traits of difficulty
	Corpus + topic model + semantic similarity info [55]	Generate-and-test	Direct: similarity measure	Offline	The input corpus	Automatic	Generic	Difficulty can be adjusted
Riddles	Database of word associations [59]	Search-based	Direct: concept relevancy	Unclear	Thesaurus Rex	Automatic	Generic	Not much
	Twitter Bot [60]	Search-based	Direct	Online: twitter bot	Uses Wikipedia	Automatic	Generic	Not much

due to the need to define structures for parallel timelines and the nonlinear causality relationships between events.

A puzzle type for which PCG has only recently been applied is programming puzzles. Dong and Barnes developed a template-based puzzle generator for an educational programming game similar to *LightBot* [61], and Valls-Vargas *et al.* [62] created a constructive generator for parallel programming puzzles based on graph grammars.

We identified four open challenges associated with procedural puzzle generation that are repeated by many of the papers we reviewed. The first challenge, in particular, for educational puzzles, such as *Refraction* and the programming puzzles, is difficulty progression, including the step-wise introduction of new concepts. Generators for these puzzles should be capable of outputing puzzles that both teach specific concepts and are of a given difficulty—the latter is harder to quantify, and achieve. As for narrative puzzles, replayability without repetition makes PCG attractive for educational puzzles.

The second challenge is developing techniques that are not narrowly focused on the rules for one specific puzzle type. Generality is difficult to obtain without sacrificing the ability of the generator to come up with novel puzzles, which is of course a core reason to use PCG in the first place: the creation of unanticipated content from a set of known inputs.

The third challenge is the assessment of quality, i.e., the development of good evaluation metrics. Metrics are puzzle-dependent and not standard, but there several common characteristics of puzzles that many researchers attempt to measure, including difficulty, variety, freshness, and aesthetics. Previous PCG surveys have also highlighted evaluation as an important area for future work [3], [6].

Aesthetic quality in itself poses the fourth challenge. We observed that procedural generation techniques for puzzles struggle to create designs that are as aesthetically pleasing as human designs.

IV. Conclusion

Given the increasing fidelity of game worlds in terms of visuals, backstory, characters, and behaviors, there is little doubt that PCG is set to become more important in games over the next years. As the critique of recent PCG-centric titles show, there are many associated challenges, but we believe that improvements in existing PCG techniques, wider adoption of PCG for new types of content (such as puzzles), as well as increased adoption of PCG across genres are unquestionably positive developments for games as a medium. They will put more creative agency into the hands of developers, allowing them to realize more ambitious projects with the same resources. However, there is also a less obvious possible benefit. By replacing the problem of creating content with the metaproblem of creating systems that create content, we obtain a deeper understanding of the essential nature of that content. Building a system that creates good puzzles requires a deeper understanding of the fundamental nature of puzzles than crafting an individual puzzle does. The same holds true for backstory, plot, and characters. Such improved understanding is likely not only to benefit games as an art form (resulting in better games and more profound experiences of playing them) but also to advance our understanding of our very human fascination with stories and games.

References

[1] A. Handy, “Interview: Markus ‘notch’ persson talks making Minecraft,” Mar. 2010. Accessed: Oct. 30, 2017. [Online.] Available: https://www.gamasutra.com/view/news/27719/Interview_Markus_Notch_Persson_Talks_Making_Minecraft

[2] D. Heaven, “When infinity gets boring: What went wrong with No Man’s Sky,” Sep. 2016. Accessed: Oct. 30, 2017. [Online.] Available: https://www.newscientist.com/article/2104873-when-infinity-gets-boring-what-went-wrong-with-no-mans-sky/

[3] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, “Procedural content generation for games: A survey,” *ACM Trans. Multimedia Comput., Commun. Appl.*, vol. 9, no. 1, 2013, Art. no. 1.

[4] S. Colton, “Automated puzzle generation,” in *Proc. AISB02 Symp. AI Creativity Arts Sci.*, 2002, pp. 99–108.

[5] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Berlin, Germany: Springer, 2016.

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, “Search-based procedural content generation: A taxonomy and survey,” *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 3, pp. 172–186, Sep. 2011.

[7] A. M. Smith and M. Mateas, “Answer set programming for procedural content generation: A design space approach,” *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 3, pp. 187–200, Sep. 2011.

[8] J. Togelius and N. Shaker, “The search-based approach,” in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Berlin, Germany: Springer, 2016, pp. 17–30.

[9] K. Compton, A. M. Smith, and M. Mateas, “Anza island: Novel gameplay using ASP,” in *Proc. 3rd Workshop Procedural Content Gener. Games*, 2012, pp. 1–13.

[10] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović, “A case study of expressively constrainable level design automation tools for a puzzle game,” in *Proc. Int. Conf. Found. Digit. Games*, 2012, pp. 156–163.

[11] “Increpare/puzzlescript: Open source html5 puzzle game engine,” 2013. [Online]. Available: https://github.com/increpare/PuzzleScript. Accessed on: Aug. 29, 2016.

[12] A. Khalifa and M. Fayek, “Automatic puzzle level generation: A general approach using a description language,” in *Proc. Comput. Creativity Games Workshop*, 2015.

[13] C.-U. Lim and D. F. Harrell, “An approach to general videogame evaluation and automatic generation using a description language,” in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1–8.

[14] J. Taylor and I. Parberry, “Procedural generation of Sokoban levels,” in *Proc. Int. North Amer. Conf. Intell. Games Simul.*, 2011, pp. 5–12.

[15] B. Kartal, N. Sohre, and S. Guy, “Generating Sokoban puzzle game levels with Monte Carlo tree search,” in *Proc. IJCAI-16 Workshop Gen. Game Playing*, 2016, pp. 47–54.

[16] J. Culberson, “Sokoban is PSPACE-complete,” in *Proc. Inform.*, 1999, vol. 4, pp. 65–76.

[17] Y. Murase, H. Matsubara, and Y. Hiraga, “Automatic making of Sokoban problems,” in *Proc. Pacific Rim Int. Conf. Artif. Intell.*, 1996, pp. 592–600.

[18] J. Taylor, T. D. Parsons, and I. Parberry, “Comparing player attention on procedurally generated vs. hand crafted Sokoban levels with an auditory stroop test,” in *Proc. Conf. Found. Digit. Games*, 2015.

[19] B. Kartal, N. Sohre, and S. J. Guy, “Data-driven Sokoban puzzle generation with Monte Carlo tree search,” in *Proc. 12th Artif. Intell. Interactive Digit. Entertainment Conf.*, 2016, pp. 58–64.

[20] N. Sturtevant, “An argument for large-scale breadth-first search for game design and content generation via a case study of fling,” in *Proc. AI Game Des. Process*, 2013, pp. 28–33.

[21] G. W. Flake and E. B. Baum, “Rush hour is PSPACE-complete, or why you should generously tip parking lot attendants,” *Theor. Comput. Sci.*, vol. 270, no. 1, pp. 895–911, 2002.

[22] S. Collette, J.-F. Raskin, and F. Servais, “On the symbolic computation of the hardest configurations of the rush hour game,” in *Proc. Int. Conf. Comput. Games*, 2006, pp. 220–233.

[23] F. Servais, “Finding hard initial configurations of rush hour with binary decision diagrams,” M.Sc. thesis, Département d’informatique, Université libre de Bruxelles, Brussels, Belgium, 2005.

[24] J. Juul, "Swap adjacent gems to make sets of three: A history of matching tile games," *Artifact*, vol. 1, no. 4, pp. 205–216, 2007.

[25] "Procedural generation of puzzle game levels - gamedev.net - your game development resource," 2014. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/game-programming/procedural-generation-of-puzzle-game-levels-r3862. Accessed on: Aug. 30, 2016.

[26] P. Song, C.-W. Fu, and D. Cohen-Or, "Recursive interlocking puzzles," *ACM Trans. Graph.*, vol. 31, no. 6, 2012, Art. no. 128.

[27] D. Ashlock, "Automatic generation of game elements via evolution," in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 289–296.

[28] N. R. Sturtevant and M. J. Ota, "Exhaustive and semi-exhaustive procedural content generation," in *Proc. 14th Artif. Intell. Interactive Digit. Entertainment Conf.*, 2018, pp. 109–115.

[29] J. Buck and J. Carter, *Mazes for Programmers: Code Your Own Twisty Little Passages*. Raleigh, NC, USA: Pragmatic Programmers, 2015.

[30] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games*. Berlin, Germany: Springer, 2016, pp. 31–55.

[31] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Trans. Comput. Intell. AI Games*, vol. 6, no. 1, pp. 78–89, Mar. 2014.

[32] M. J. Nelson and A. M. Smith, "ASP with applications to mazes and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Berlin, Germany: Springer, 2016, pp. 143–157.

[33] N. Brewer, "Computerized dungeons and randomly generated worlds: From Rogue to Minecraft," *Proc. IEEE*, vol. 105, no. 5, pp. 970–977, May 2017.

[34] A. M. Smith, E. Butler, and Z. Popovic, "Quantifying over play: Constraining undesirable solutions in puzzle design," in *Proc. 8th Int. Conf. Found. Digit. Games*, 2013, pp. 221–228.

[35] D. Williams-King, J. Denzinger, J. Aycock, and B. Stephenson, "The gold standard: Automatically generating puzzle game levels," in *Proc. 8th Artif. Intell. Interactive Digit Entertainment Conf.*, 2012, pp. 191–196.

[36] J. Dormans, "Adventures in level design: Generating missions and spaces for action adventure games," in *Proc. Workshop Procedural Content Gener. Games*, 2010, Art. no. 1.

[37] E. Butler, A. M. Smith, Y.-E. Liu, and Z. Popovic, "A mixed-initiative tool for designing level progressions in games," in *Proc. 26th Annu. ACM Symp. User Interface Softw. Technol.*, 2013, pp. 377–386.

[38] C. Fernández-Vara and A. Thomson, "Procedural generation of narrative puzzles in adventure games: The puzzle-dice system," in *Proc. 3rd Workshop Procedural Content Gener. Games*, 2012, pp. 12–17.

[39] M. Treanor *et al.*, "Playable experiences at AIIDE 2017," in *Proc. 13th AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, Snowbird, UT, USA, Oct., 2017.

[40] I. Dart and M. J. Nelson, "Smart terrain causality chains for adventure-game puzzle generation," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 328–334.

[41] J. Doran and I. Parberry, "A prototype quest generator based on a structural analysis of quests from four MMORPGs," in *Proc. 2nd Int. Workshop Procedural Content Gener. Games*, 2011, Art. no. 1.

[42] A. Isaksen, D. Wallace, A. Finkelstein, and A. Nealen, "Simulating strategy and dexterity for puzzle games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2017, pp. 142–149.

[43] M. Shaker, M. H. Sarhan, O. Al Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2013, pp. 1–8.

[44] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *Proc. 9th AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, 2013, pp. 72–78.

[45] M. Shaker, N. Shaker, J. Togelius, and M. Abou-Zleikha, "A progressive approach to content generation," in *Proc. Eur. Conf. Appl. Evol. Comput.*, 2015, pp. 381–393.

[46] L. Ferreira and C. Toledo, "A search-based approach for generating angry birds levels," in *Proc. IEEE Conf. Comput. Intell. Games.*, 2014, pp. 1–8.

[47] M. Stephenson and J. Renz, "Procedural generation of levels for angry birds style physics games," in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, 2016, pp. 225–231.

[48] L. Carroll, *The Game of Logic*, vol. 1. Alexandria, Egypt: Library Alexandria, 1886.

[49] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," in *Proc. IEEE Congr. Evol. Comput.*, 2007, pp. 1382–1389.

[50] T. Boothby, L. Svec, and T. Zhang, "Generating Sudoku puzzles as an inverse problem," *Math. Contest Model.*, vol. 24, 2008.

[51] Y.-H. Xue, B.-B. Jiang, Y. Li, G.-F. Yan, and H.-F. Sun, "Sudoku puzzles generating: From easy to evil," *Math. Pract. Theory*, vol. 21, pp. 1–7, 2009.

[52] M. Gebser, "gen_sudoku.gringo," Source Code, Mar. 2010. Accessed: Oct. 13, 2017. [Online]. Available: https://asparagus.cs.uni-potsdam.de/encoding/show/id/12739

[53] B. Fatemi, S. M. Kazemi, and N. Mehrasa, "Rating and generating Sudoku puzzles based on constraint satisfaction problems," *World Acad. Sci., Eng. Technol., Int. J. Comput., Elect., Autom., Control, Inf. Eng.*, vol. 8, no. 10, pp. 1811–1816, 2014.

[54] D. Oranchak, "Evolutionary algorithm for generation of entertaining Shinro logic puzzles," in *Proc. Eur. Conf. Appl. Evol. Comput.*, 2010, pp. 181–190.

[55] B. Pintér, G. Voros, Z. Szabó, and A. Lorincz, "Automated word puzzle generation via topic dictionaries," in *Proc. Int. Conf. Mach. Learn. — Sparsity Dictionaries Projections Mach. Learn. Signal Process. Workshop*, Edinburgh, Scotland, Jun. 30, 2012.

[56] S. Colton, "Automated theory formation in pure mathematics," Ph.D. dissertation, Dept. Artif. Intell., The University of Edinburgh, Edinburgh, U.K., 2001.

[57] L. Rigutini, M. Diligenti, M. Maggini, and M. Gori, "A fully automatic crossword generator," in *Proc. 7th Int. Conf. Mach. Learn. Appl.*, 2008, pp. 362–367.

[58] J. R. R. Tolkien, *The Hobbit, or There and Back Again*. London, U.K.: G. Allen, 1937.

[59] P. Galván, V. Francisco, R. Hervás, and G. Méndez, "Riddle generation using word associations," in *Proc. Int. Conf. Lang. Resources Eval.*, 2016, pp. 2407–2412.

[60] I. Guerrero, B. Verhoeven, F. Barbieri, P. Martins, and R. Pérez y Pérez, "TheRiddlerBot: A next step on the ladder towards computational creativity," in *Proc. 6th Int. Conf. Comput. Creativity*, 2015, pp. 315–322.

[61] Y. Dong and T. Barnes, "Evaluation of a template-based puzzle generator for an educational programming game," in *Proc. 12th Int. Conf. Found. Digit. Games*, 2017, Art. no. 40.

[62] J. Valls-Vargas, J. Zhu, and S. Ontañón, "Graph grammar-based controllable generation of puzzles for a learning game about parallel programming," in *Proc. 12th Int. Conf. Found. Digit. Games*, 2017, Art. no. 7.

Barbara De Kegel received the B.Sc. degree in computer science from University College Dublin (UCD), Dublin, Ireland, in 2015, and the M.Sc. degree in computer science, specialized in interactive entertainment technology, from Trinity College Dublin, Dublin, Ireland, in 2016. She is currently working toward the Ph.D. degree in computational cancer biology at Systems Biology Ireland, UCD.

She was a Software Engineer with Havok/Microsoft, Dublin, Ireland, from 2016 to 2018.

Mads Haahr (S'01–M'03) received the B.Sc. and M.Sc. degrees in computer science and English from the University of Copenhagen, Copenhagen, Denmark, in 1996 and 1999, respectively, and the Ph.D. degree in computer science from Trinity College Dublin, Dublin, Ireland, in 2004.

He is currently a Faculty and serves as the Course Director for the M.Sc. in Interactive Digital Media with Trinity College Dublin. He has authored/coauthored more than 80 peer-reviewed publications and founded the award-winning game studio Haunted Planet Studios in 2010. He is also known for creating the Internet's premier true random number service RANDOM.ORG in 1998.