Design-Centric Maze Generation

Paul Hyunjin Kim
The Ohio State University
Columbus, Ohio
kim.3983@osu.edu

Jacob Grove
The Ohio State University
Columbus, Ohio
grove.217@osu.edu

Skylar Wurster
The Ohio State University
Columbus, Ohio
wurster.18@osu.edu

Roger Crawfis
The Ohio State University
Columbus, Ohio
crawfis.3@osu.edu

ABSTRACT

A maze is a common structure in a game level. When we design game levels having a different purpose of each level, we may desire mazes with different topological properties, such as lots of branches or long straight-ways. Thus, we need the ability to design mazes based on our game mechanics. In this paper, we introduce our design-centric maze generation in which designers can input their desired properties to create their own mazes. Our method also enables the designers to control the topology of the solution path of a maze. Additionally, this method can provide several mazes which satisfy the given desired properties allowing designers to choose the best maze and use it to build game content for a game level. To demonstrate how useful our design-centric method is, this paper provides several use-cases of building actual game levels and shows that we can design the levels effectively using our method.

CCS CONCEPTS

• **Computing methodologies** → Graphics input devices;

KEYWORDS

Perfect Maze, Maze Generation, Topological Properties of Maze, Search-Based Procedural Content Generation

ACM Reference Format:
Paul Hyunjin Kim, Jacob Grove, Skylar Wurster, and Roger Crawfis. 2019. Design-Centric Maze Generation. In *The Fourteenth International Conference on the Foundations of Digital Games (FDG'19), August 26–30, 2019, San Luis Obispo, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3337722.3341854

1 INTRODUCTION

A maze is a puzzle where a player finds a path from the starting point to the ending point. Nowadays, mazes are also used as a platform for a computer game level. Random mazes can be used to design game levels, but in some cases mazes with specific properties may be desired to cause an effect on the player. For example, when we design a level for exploring dungeons, we may desire a relatively high number of branches in the maze to disorient the player. Also, when we design a level in which the player is being chased, we may desire long straight-ways on a maze to make a player run at high speeds. To create a maze with desired properties, we can use existing computer maze generation tools like [1], but they have a lack of control over maze generation. For example, we can control only the size of the maze and perhaps the random seed for the generation. In our research, we developed a design-centric maze generation method that allows users to input their own desired topological properties. We have detailed control over the maze's topology and also the solution path of the maze. Several mazes satisfying the given desired properties can be returned as an output of this method so that designers can choose the best one that fits their use. Once the best maze is chosen, designers can perform post-processing, such as building an actual game level, on the maze. To prove the usefulness of our method, we demonstrate the performance of generating desired mazes and provide several use-cases of designing actual computer game levels.

2 RELATED WORKS

There are many methods focusing on solving a maze, but this paper focuses on generating a maze. In the real world, many mazes are created manually. We can easily see lots of maze puzzle books, where mazes are drawn by hand. In [5], Since 2002, a maze artist Christopher Berg shows his astonishing maze works created manually on paper. In [10] and [20], we can also see many life-size maze attractions designed and constructed by maze design companies. Berg said that this manual creation process requires many hours of practice to design paths with complicated structure. Thus, drawing a maze is not easy for people with less experience.

There are many computer algorithms to draw a maze automatically. In [21] and [8], a comprehensive list of maze generation algorithms is provided. Most of these are based on spanning tree algorithms. As explained in [21] and [8], classic spanning tree algorithms, such as Prim's algorithm, are used as the maze generation algorithm. In [7], the maze generation process of the algorithm is animated step by step so that we can easily see how they proceed. The RRT Page [15] also shows mazes created using rapidly-exploring random trees.

Research has been conducted on developing the aesthetic aspects of a maze. Xu et al. [25] constructed a maze with obfuscated

FDG'19, August 26–30, 2019, San Luis Obispo, CA, USA

ACM ISBN 978-1-4503-7217-6/19/08…$15.00
https://doi.org/10.1145/3337722.3341854

structure by adopting a vortex shape on it. Also, Pedersen et al. [19] created organic labyrinths, which resemble input images, by evolving simple curves. In [26] and [24], a maze was generated based on a given image. Consequently, the input image is filled with paths of a maze. In [9], paths of a maze were decorated with textures designed by artists. In [18], Okamoto et al. showed work to create a picturesque maze. Picturesque mazes look like a normal maze, but the filled solution path reveals an image.

Research has also been conducted on controlling the topology of generated mazes to some user specifications. Ashlock et al. [4] developed an evolutionary algorithm, in which a maze-like level is evolved with dynamic programming to control the resulting structure. In this research, depending on the type of level representation, the generator has different tendencies over generated level structures. Bosch et al. [6] proposed a method that utilizes integer program to generate a labyrinth with path-segment tiles. In the integer program, we can input some constraints to force the labyrinth to be as symmetric as possible. Kaplan [11] did some research regarding a reconfigurable maze, where rotating maze parts yields a maze with another topology. In [16], Maung et al. used array grammar rules to define a path structure, and a maze was generated based on the rules. Kim et al. [13] also provided a method that applies existing spanning tree algorithms. When a user has desired topological properties, this method attempts to choose a spanning tree algorithm that is best for creating mazes with the desired properties effectively. In [14], Kim et al. proposes a maze generation method that applies a machine learning technique so that mazes with any valid topological constraints can be obtained successfully.

3 RESEARCH DOMAIN

Our research domain is a rectangular perfect maze as shown in Figure 1. A perfect maze is a type of maze that has no loops and no inaccessible areas. Since the maze has the same properties as a spanning tree, we can consider a rectangular perfect maze as a spanning tree in a rectangular grid. In our research, an MxN rectangular grid indicates an arrangement of rectangular cells in M columns and N rows. We consider each cell center as a node and lines between cell centers as an edge. Each node has neighboring edges in either vertical or horizontal directions. A MxN grid has $M(N-1)+(M-1)N$ edges in total. For any spanning tree on a rectangular grid, the edges of the spanning tree correspond to paths of a perfect maze. Thus, to generate a maze in a grid, we can use existing minimum spanning tree algorithms, such as Kruskal's algorithm with randomized weights. Since there is a duality between a perfect maze and a spanning tree, we will use both terminologies interchangeably in this paper. The difference is that a maze has a start and end cell.

To explain more about our research domain, the perfect maze, we introduce several components that we can have in a perfect maze. Here, some descriptions are given from the viewpoint of a player playing a maze puzzle.

(1) Starting Point and Ending Point: The starting point is a point where a player starts to solve a maze. The ending point is a point where the player needs to arrive to finish the maze puzzle.
(2) Solution Path: The solution path is a path between the start point and the end point. Since a perfect maze has no loop, there is a unique solution path.
(3) Dead-End Trees: All paths on a perfect maze which are not the solution path are dead-end trees. One maze can have several dead-end trees, and they branch from the solution path. In our research, we define three types of dead-end trees.
    - Forward Dead-End: The forward dead-end is a dead-end tree that heads towards the ending point. It provides the illusion that the dead-end tree is leading to the ending point.
    - Backward Dead-End: The backward dead-end is a dead-end tree that turns away from the ending point. It has opposite directional properties compared to the forward dead-end.
    - Alcove: An alcove is a dead-end which is only a straight path.

    More descriptions about these three types of dead-end trees can be found in [12].
(4) Maze Cells: A maze cell is a unit of a perfect maze, and there can be five types of maze cells based on the shape of edges on its center. Each type of maze cell is illustrated in Figure 2, and the description of each type is given below.
    - Turn: On a perfect maze, a turn cell consists of a single vertical path and a single horizontal path. There are four possible turn cells as shown in Figure 2(a).
    - Straight: Straight cells have two paths that are either both horizontal or both vertical. There are two types of straight cells: a vertical straight cell, in which both paths have vertical directions; and a horizontal straight cell, in which both paths have horizontal directions.
    - T-Junction: A T-junction cell has has exactly 3 edge from the cell center and represents a decision on a perfect maze. It has the shape of a T. There are four shapes of T-junctions, as shown in Figure 2(c). In a perfect maze, when a player enters a T-junction, the player needs to decide which path to take.
    - Cross-Junction: A cross-junction cell has all four paths out of the cell center. The player has three choices of direction to decide from in a perfect maze.
    - Terminal: A terminal cell has only one path on the cell center. A player needs to go back when the player enters the terminal cell. There are four shapes of terminal cells as depicted in Figure 2(e).

Note that in this paper, we focus only on the topology and not art design or game mechanics. A terminal cell may be a large organic room, and a cross-junction cell may require a specific game mechanic such as jumping off a bridge.

4 MOTIVATION

In this section, we provide more detailed examples of designing a game level using a maze.

Example 1: Consider designing a level with a search quest. As described in [2], the search quest is a level archetype where a player is asked to search to collect required items. We design a game level

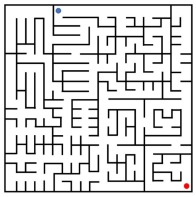

Figure 1: An example of a rectangular perfect maze with the starting point (blue) and the ending point (red).

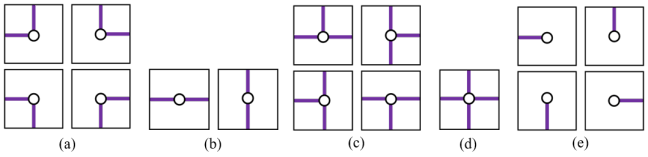

Figure 2: Representation of (a) turn cells, (b) straight cells, (c) T-junction cells, (d) cross-junction cells, and (e) terminal cells. Purple lines and black circles denote a path and a cell center, respectively.

such that the player needs to spend lots of time finding paths to each quest item. Thus, we might need a level structure that is complicated to disorient the player. In this design, we can use a maze with a relatively higher number of junctions so that the level has enough places where a decision is necessary, leading to more time spent searching.

Example 2: Assume that we are designing a game level where a player is fleeing from enemies that chase the player. To give this game level more tension, we can design the game level using a single path, such as a labyrinth. Then, the player is forced to look ahead and has to deal with any events on the single path to the exit. By adding a few short dead-ends on the path, we can give the player little rooms to get items that help the player flee better, such as a run speed increase. But the player still has less control over evading the enemy and has to rely on running as fast as possible to escape. In this design, we can use a maze with a few numbers of junctions and long straight-ways.

In these examples, we can see that mazes with different topological properties can help in designing the corresponding levels. To create the desired mazes using the current maze generation tools that exist, we would need to generate random mazes repeatedly until we get one close enough to the desired maze. It is time-consuming to find such desired mazes. Therefore, in this paper, we introduce our design-centric method, which can help users obtain these mazes more effectively.

5 DEFINING DESIRED PROPERTIES

In this section, we introduce the desired properties that we can define over a maze.

5.1 Basic Properties

First, we define some basic properties of the maze and its solution path. These are quantitative attributes of a perfect maze, and are referred to as maze metrics.

Solution Path

- Starting Point and Ending Point
- #Turns: The number or percentage of turn cells on the solution path.
- #Straights: The number or percentage of straight cells on the solution path.
- #Decisions: The number or percentage of decisions on the solution path. Each decision will be either a T-junction cell or a cross-junction cell.
- SolutionPathLength: The number or percentage of edges on the solution path. Designer-friendly semantics can also be used such as short, long, and very long.

Maze

- Size: Denotes the size of a maze. It will be defined by the number of columns and the number of rows of a grid.
- #Turns: Denotes the number or percentage of turn cells in the maze.
- #Straights: Denotes the number or percentage of straight cells in the maze.
- #T-Junctions: Denotes the number or percentage of T-junction cells in the maze.
- #Cross-Junctions: Denotes the number or percentage of cross-junction cells in the maze.
- #Terminals: Denotes the number or percentage of terminal cells in the maze.

5.2 Higher-Level Properties

Additionally, we can define higher-level topological properties on a maze such as in [21]. These can be more intuitively related to design-centric properties than the above basic properties. Here, we provide a list of higher-level properties.

- Run: Indicates how long a maze path is straight before it meets a turn or junction. A higher run will give us mazes with long straight-ways. A lower run will give us mazes with short straight-ways with lots of turns and/or branches.
- River Factor: Indicates relative density of dead-ends and junctions in the maze. A Low river factor indicates that there are many short dead-ends on each dead-end tree. A high river factor indicates that there are a few long dead-ends on each dead-end tree.
- Bias: Tendency to have straight-ways in one direction more often than another direction. For example, a maze with a bias will have more vertical straight-ways then horizontal straight-ways.
- Agility Versus Speed (AVS): Indicates the twistiness of a path of a maze. When we have a high AVS, we may have many consecutive turns on a path so that a player is required to be quick and nimble, changing direction often. In contrast, when we have a low AVS, we may have long straight-ways on a path so that a player needs less control, but may pick up speed.
- Homogeneity: Indicates the amount of homogeneity over a maze properties. High homogeneity yields a maze with homogeneous properties, but low homogeneity gives a maze with diverse properties.

- Hidden Factor: Indicates how hard it is to recognize the shape of paths from any place while playing a maze. a high hidden factor means the player will be easily lost and confused, and a lower hidden factor means the player may easily find the path to the exit.

6 DESIGN-CENTRIC METHOD

In this section, we provide an overview of our design-centric maze generation method. Our method consists of four stages, the input stage, solution path generation stage, maze generation stage, and output stage. In this overview, we explain what is done in each stage briefly.

6.1 Input Stage

In this stage, we input the desired properties of a maze. The desired properties are defined by properties explained in Section 5. Higher-lvel properties are converte to basic properties.

6.2 Solution Path Generation Stage

In this stage, we generate the solution path satisfying the given desired properties. If there is no specified starting point and ending point, we assume the top-left corner and bottom-right corner of the grid, respectively. More descriptions of this path generation will be given in Section 7.

6.3 Maze Generation Stage

In this stage, we generate a maze satisfying the given desired properties. When we generated the desired solution path in the previous stage, we need to create a maze containing the path. For this, we can generate a maze first and check whether the maze has the desired solution path. However, this technique will have a very low probability to find mazes that satisfy the desired properties and the generated path at the same time. Instead, we insert the generated path on a grid as a hard constraint. As described in Section 8, our maze generation method allows hard constraint inputs on the grid such that the resulting generated maze is forced to contain the constraints in its topology. Thus, by giving the desired path as a hard constraint, we are guaranteed to have the path in the resulting maze. More descriptions of this maze generation will be given in Section 7.

6.4 Output Stage

In this stage, we provide a single maze as an output. Alternatively, we can provide a set of mazes as an output - all which satisfy the given constraints - so that a user can select among them or design multiple levels. From there, designers may iterate on the resulting maze and introduce loops or empty areas to diversify gameplay.

7 SOLUTION PATH AND MAZE GENERATION WITH DESIRED PROPERTIES

In this section, we explain briefly how we create the solution path and a maze satisfying the given topological properties. To the best of our knowledge, it is not easy to generate the desired path or the desired maze directly. Creating a desired maze with a desired solution path is even harder. Instead, in our research, we apply an approach called search-based procedural content generation[23]. This section explains this approach first and then the path generation and maze generation.

7.1 Search-Based Procedural Content Generation (SBPCG)

SBPCG is one approach to generate the desired content using a searching mechanism. When we search for the desired content, we iteratively generate and test results to see whether the generated result has properties close to the desired properties. We can continuously generate content until we have the desired one. However, the process is not guaranteed to finish, especially when we have a very low probability to find a desired one. To fix this, another termination protocol may be introduced to the searching process. We can iteratively generate a fixed number of results and return one closest to the desired one. This guarantees that the searching process will terminate, but now the resulting content might not have the desired properties.

7.2 SBPCG-Based Solution Path Generation

Here, we explain how we accelerate the SBPCG approach to make it feasible, as the space of perfect mazes is very large [14]. To find the desired path, we need to know how to generate a path on a grid. One option is to apply a random walk algorithm. However, the algorithm takes a very long time to generate a path and is not guaranteed to terminate within some amount of time. Instead, we can use pathfinding algorithms such as randomized depth-first search which visits neighboring nodes of the current node randomly. These algorithms introduce bias, however, and tend to generate paths with certain topological characteristics, such as long straightways.

We developed a new efficient path generation and enumeration algorithm. This method generates edges column by column in a grid. In Figure 3, we see that each column consists of different edges on a grid. In Figure 4, the path generation starts from the leftmost column and ends at the rightmost column. On each column, we generate edges that do not give us a loop or cause a disconnect with the previously generated edges so that a valid path is maintained. In our method, we set up several rules to create valid edges on each column rapidly without looking at the previously generated edges. This method performs quickly and generates path with uniform distributions, unlike randomized depth-first search. More details of this path generation will be given in the forthcoming report.

Therefore, when we iteratively generate paths using the SBPCG approach to find the desired path, we could create paths using our new method. However, since random generation can repeatedly give us duplicated paths, it can be rather inefficient for finding a desired path. We could enumerate all paths using this method, but enumeration is an NP problem. This means that it can take a long time to search over all possible paths. As an optimization, we can apply some constraints over the path enumeration to reduce the search time. Assume that we have the desired length (desired number of edges) of a path where the two end points are the top-left node and the bottom-right node in a grid. When we generate the edges of a path on one column, we count how many edges we have generated between the leftmost column and the current column.

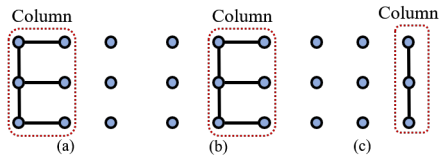

Figure 3: Figures show corresponding edges of each column on a 3x3 grid. (a) Edges of the leftmost column. (b) Edges of the second left column. (c) Edges of the rightmost column. Note that the rightmost column has only vertical edges.

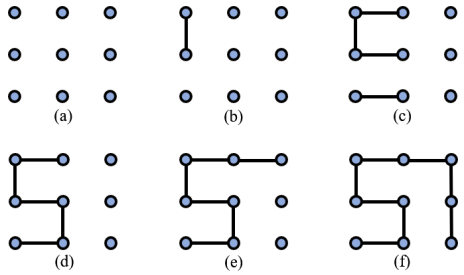

Figure 4: Illustration of the column-based path generation on a 3x3 grid.

If the number of edges already exceeds a valid amount, we stop. These applied constraints can prevent the enumeration process from spending time on generating non-desired paths.

7.3 SBPCG-Based Maze Generation

Instead, we can apply spanning tree enumeration to all mazes for the SBPCG approach, meaning a desired maze will be found eventuall. However, the enumerated space of the spanning tree is huge. For example, in a 6x6 grid, there are trillions of spanning trees. Searching through trillions of spanning trees for a desired maze is not feasible.

In our research, we developed a new maze generation method for finding the desired mazes. In this method, we pre-build a set of machine learning models that are trained to generate spanning trees with specific topological properties. To train the models effectively, this method uses the enumerated spanning tree data. Based on the given desired properties, we select a proper model in the set of the trained models and use it to generate spanning trees for the SBPCG approach. This machine learning-based method returns desired mazes in a short amount of time. More details of this method including the training process are found in [14].

In the learning-based method, we need the enumerated spanning tree data to train the models. When the grid size gets larger, the size of the enumerated data gets exponentially larger. We avoid enumerating other larger grid domains for training data, and instead implement a new technique for grids larger than 6x6. In a large grid, our new maze generation works on a hierarchical maze, as shown in Figure 5, where we combine small mazes to create a large maze.

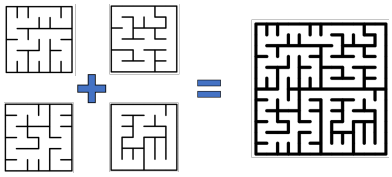

Figure 5: Overview of hierarchical maze construction. 6x6 small mazes are stitched to build 12x12 large maze.

8 APPLYING HARD CONSTRAINTS

When we define desired properties in our method, we can give hard constraints along with the desired properties. In our method, we can specify some edges of the solution path directly over a grid. Then, we can enumerate the results that pass through the specified edges. Likewise, we can specify cells of a maze directly over a grid. Through these hard constraints, maze designers can have direct control over maze topology.

9 MAZE DESIGN TOOL

Based on our design-centric method, we have developed a maze design tool as shown in Figure 7. In the tool, we can specify several basic properties as input parameters, such as the size of a maze, length of the solution path, and properties of a solution path and maze. For a hierarchical maze, in addition to setting up metric values for a large maze, we can specify metric values for the stitched small mazes locally.

In this tool, we also let users input some higher-level properties that can be defined quantitatively using maze properties, such as a river factor and AVS. Then, inside the tool, the input properties are converted to values of maze metrics, and desired mazes are found based on those metric values. For example, a high AVS is converted to a large value of #Turns and a small value of #Straights. Also, a high river factor is converted to a low value of #T-Junctions and a relatively higher value of #Straights. However, properties like the hidden factor and homogeneity have some cognitive term, and are difficult to define quantitatively with the basic metrics. Designers may have a different definition for these subjective properties. Thus, in our tool, we let the designers use basic properties to define these higher-level properties by themselves.

Hard constraints also can be specified in the tool, as mentioned in Section 8 so that we can obtain the maze satisfying the input constraints.

After input parameters are specified, our tool generates a fixed number of mazes based on the input and returns the best maze among those results to a user. If a user does not like the result, he or she can obtain desired mazes with different topologies by simply clicking the 'Create Maze' button several times.

Now, in the next section, we represent several use-cases of designing game levels to show how our method can be helpful in designing actual game content. In these use-cases, we used our design tool described in Section 9 to generate desired mazes. Note that when higher-level design-centric properties are used in the use-cases, some calculation was done to turn those into to the basic metric values defined.

10 RESULTS

In this section, to validate the effectiveness of our design-centric method, we compare the expressive range [22] of our method with the expressive range of a previously developed method [13], which chooses the best spanning tree algorithm to find desired mazes. Additionally, we provide several use-cases of designing game levels and show how our method builds the desired game levels by generating the corresponding mazes.

First, we obtain expressive range data, for a 18x18 maze, using 1,000 mazes. In Figure 6, the expressive ranges are visualized using 2D histograms with contour maps. In the histograms, the color darkness corresponds to the number of mazes that have the associated metrics. A darker color means that more mazes belong to that space. Dotted red lines are used to show whether expressive ranges (generative spaces) cover mazes of the corresponding properties. Note, in this application, we desire a small expressive range near the target.

Figure 6 shows that we can have direct control over resulting topologies using our method. While the choice-based method does not have expressive ranges covering input metric values, our method has expressive ranges covering input parameter values. It indicates that maze generation can be controlled well by our method so that we can generate desired mazes effectively.

Rolling Ball Game: Suppose that we design a level for a rolling ball game. In the rolling ball game, we have one ball and tilt a platform to move the ball toward the goal point. This game type was also applied in the commercial game "The Legend of Zelda: Breath of the Wild"[17] as shown in Figure 8.

In the game level, it is important to give different levels of difficulty over the platform so that a player can have various tensions during the gameplay. To manipulate the difficulty, we can use the agility vs. speed (AVS) concept explained in Section 5 on the platform. When we have a low AVS, the level will have long straight-ways so that we tilt the platform in one direction for a long time. When we have a high AVS, the level will have curving passages so that we need to change the direction of tilt frequently.

In this design, we specified different amounts of AVS for each quadrant of a 12x12 maze so that we can have various difficulties while we play the game in this 12x12 maze. For a high AVS, we defined a high number of turns and a low number of straights as input parameters. Likewise, for a low AVS, we gave a low number of turns and a high number of straights. Figure 9(a) shows a set of 12x12 mazes we obtained using our tool. We specified different AVS properties locally; top-left and bottom-right quadrants have a low AVS, and top-right and bottom-left quadrants have a high AVS. Resulting game level is shown in Figure 9(b). The goal point on the level is denoted by a flag.

Plumbing Game: Suppose that we design a plumbing game, as shown in Figure 10. The plumbing game is a puzzle where we need to rotate pipe parts to assemble a pipe system that connects the starting point(s) and the ending point(s).

The most important design factor for designing a puzzle for this type of game is that the puzzle is actually solvable. To create a solvable plumbing puzzle, we generate a solution path using our method and place the corresponding pipe part on each maze cell. Then, we rotate each pipe part randomly. We can make a level more difficult by adding junctions and filling out the maze. As we add more and more junctions to the maze, the player will have to make more choices about which way to rotate each pipe.

Using our method, we generate a difficult plumbing puzzle. We input a relatively higher number of T-junctions for a 6x6 grid. In Figure 11(a), we can see a set of mazes generated by our tool. Figure 11(b) shows a pipe system where pipe parts are placed on one of the mazes in Figure 11(a). In Figure 11(c), an unsolved puzzle level is obtained by rotating the pipe parts.

Running Game: Nowadays, as represented in [3], there are some running games which utilize a treadmill as an input device. A player can use the various system to watch the game scene, such as a tablet or a VR system. In the game, the player needs to run on the treadmill to move a character in the game.

An important design factor for this kind of a game is to not make a player bored during the gameplay. Thus, besides creating a path for running, we need to add some natural looking scenery around the path. To create a path in the game, we can use the solution path generation ability in our method. Also, to add natural looking scenery, we can use the maze obtained by our method to build some road network around the path. Then we may place ambient objects, such as trees, rocks, and other models, around the path. The game might give more freedom to a player so that the player can also explore the surrounding road map during the gameplay.

In this design, we generate a path for running over a 12x6 maze first. When we specify the path, we select a somewhat average SolutionPathLength so that we can have enough room for the surrounding road map over a maze. To create a 12x6 maze using our method, we stitch two 6x6 mazes side by side. We used a hard constraint to have the path go through the bottom middle between the mazes. Then, we ask for a relatively higher number of T-junctions over the maze topology to have a road network with junctions around the running path. Figure 12(a) shows a set of mazes generated by our tool. In this set, we chose one maze and, as shown in Figure 12(b), built a level structure based on the maze. In Figure 12(c), we can see the actual gameplay scene with the marked running path.

Table 1 provides the performance of our method in designing game levels of the above use-cases. In Table 1, we can see that our method needs only a few input parameters to build the desired levels. Table 1 also shows how accurately our method generates desired mazes. In Section 9, we mentioned that our tool generates 1,000 mazes to find the desired ones. To demonstrate the accuracy of our method, we chose the top 10 mazes among the 1,000 mazes and calculated average distances between a vector of input desired parameters and vectors of parameters measured from the top 10 mazes. When the average distance value (ADV) is close to 0, it means that our method has more accuracy in finding desired mazes. As shown in Table 1, ADVs for all use-cases are 0, which means that the top 10 mazes generated with our method exactly match the desired properties given. Additionally, we can see the total time to generate 1,000 mazes using our tool. As described in Table 1, our method generates 1,000 mazes in a reasonable amount of time, which is usually less than 1 second for each use-case. This is a vast improvement over other approaches for designing maze-based game levels. Then, since we can make various maze structures with

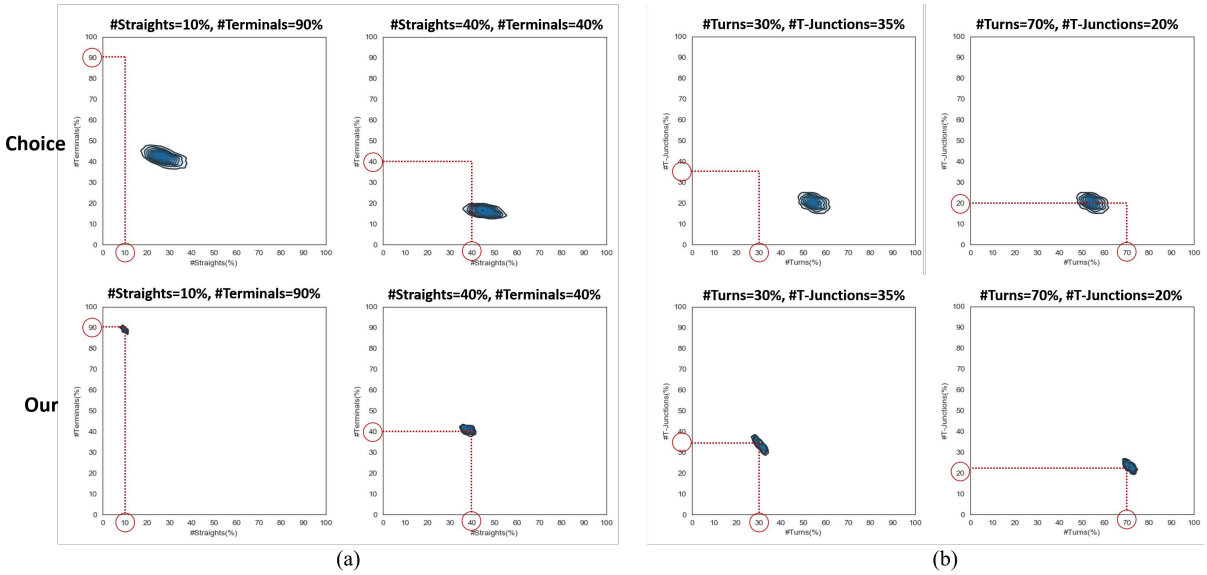

Figure 6: Charts showing expressive ranges in a 18x18 grid regarding 2D basic parameters ((a) #Straights & #Terminals, (b)#Turns & #T-Junctions). In each panel, charts in the top row correspond to results of the choice-based method, and our method is sown in the bottom row. Charts in each column of each panel show histograms of the same input metric values. Darker color means that there more mazes with the associated metrics we are generated. Dotted red lines are used to show the desired metric values.

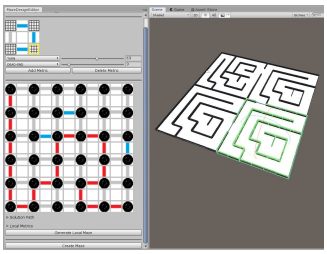

Figure 7: Figure showing a maze design tool we have developed.

Figure 8: Figure showing one scene of the commercial game "The Legend of Zelda: Breadth of the Wild"[17] that contains a maze-based rolling ball game.

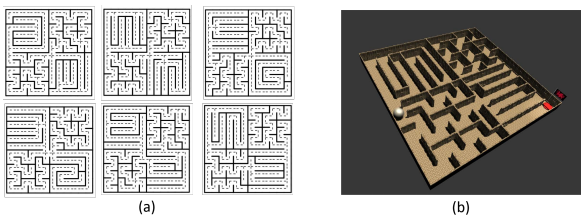

Figure 9: Figures showing (a) a set of mazes created by our tool for the rolling ball game and (b) an actual gameplay view. In (a), each quadrant of the maze was given different desired properties. In (b), the goal position is marked by a flag, so we need to roll the silver ball toward the flag.

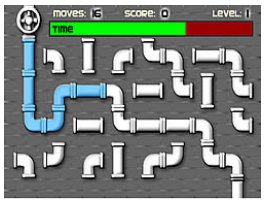

Figure 10: Example showing the plumbing game captured from https://ko.y8.com/games/plumber_game

little difficulty, we allow designers to find the kinds of features they're looking for.

11 CONCLUSION AND FUTURE WORK

In this paper, we introduced our design-centric maze generation method. In our method, we can give desired properties for the

Game	Input Parameters	Error (ADV)	Time to Generate 1,000 Mazes
Rolling Ball	*Top-Left & Bottom-Right Mazes:* #Turns=15%, #Straights=85% *Top-Right & Bottom-Left Mazes:* #Turns=85%, #Straights=15%	0.0	0.54s
Plumbing	#T-Junctions=85%	0.0	0.13s
Running	SolutionPathLength=50%, #T-Junctions=75%	0.0	0.51s

Table 1: Table showing performance of our method in designing game level for each use-case

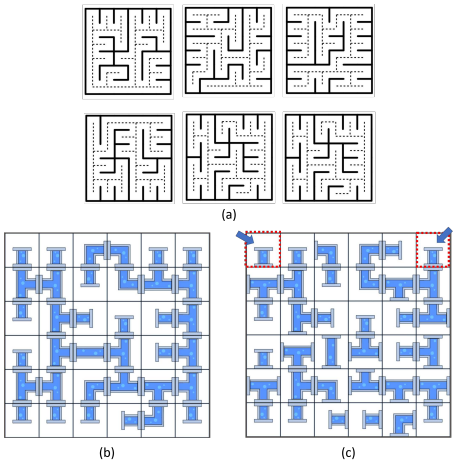

Figure 11: (a) A set of mazes obtained by our tool, (b) a solved plumbing game, and (c) Puzzle where we need to connect the pipe parts between the ending parts denoted by arrows.

solution path and metrics for the whole maze topology which are used as inputs to generate a maze. Using the SBPCG approach, we generate the desired maze satisfying both the desired solution path and design-based parameters & constraints. We quickly compute desired mazes, allowing users to obtain the maze that is best for them. To demonstrate the effectiveness of our method, we provided several use-cases of building an actual game level and see how our method could be used to help users design their desired game levels. We observed that our method generates the desired levels effectively with a few input parameters. This design-centric method has a promising potential to be applied to design content in various fields, especially the field of computer game content generation.

In the future, we will ask expert level designers and even novices to use our maze-design tool to design their own levels and investigate how our tool works. We expect to have valuable feedback from them so that we can improve our tool to be more designer-friendly.

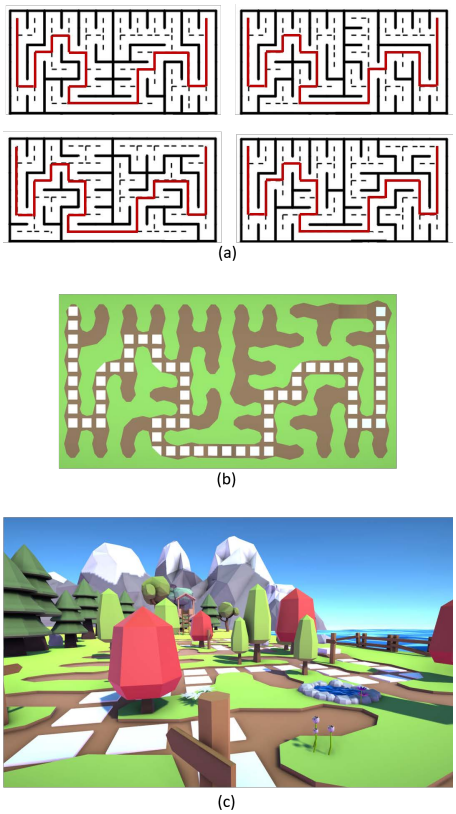

Figure 12: (a) A set of mazes generated for the running game, (b) a level structure built based on one of the mazes in the set, and (c) an actual gameplay level. The running path is denoted by red lines in (a) and by dotted-lines in (b).

REFERENCES

[1] *Maze Generator.* http://www.mazegenerator.net.
[2] *RPG Design Patterns.* https://rpgpatterns.soe.ucsc.edu/doku.php.
[3] *Zwift Run.* https://zwift.com/en/run/.
[4] D. Ashlock, C. Lee, and C. McGuinness. 2011. Search-Based Procedural Generation of Maze-Like Levels. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (September 2011), 260–273. DOI:http://dx.doi.org/10.1109/TCIAIG.2011.2138707
[5] Christopher Berg. *Amazeing Art.* http://amazeingart.com.
[6] Robert Bosch, Sarah Fries, Mäneka Puligandl, and Karen Ressler. 2013. From Path-Segment Tiles to Loops and Labyrinths. In *Proceedings of Bridges 2013: Mathematics, Music, Art, Architecture, Culture.*
[7] Jamis Buck. *HTML 5 Presentation with Demos of Maze Generation Algorithms.* www.jamisbuck.org/presentations/rubyconf2011/index.html.
[8] Jamis Buck. 2015. *Mazes for Programmers: Code Your Own Twisty Little Passages.* Pragmatic Bookshelf.

[9] Wen-Shou Chou. 2016. Rectangular Maze Construction by Combining Algorithms and Designed Graph Patterns. *GSTF Journal on Computing (JOC)* (August 2016).

[10] Adrian Fisher. *Maze Maker.* http://mazemaker.com.

[11] Craig S. Kaplan. 2014. The Design of a Reconfigurable Maze. In *Proceedings of Bridges 2014: Mathematics, Music, Art, Architecture, Culture.*

[12] P. H. Kim and R. Crawfis. 2015. The quest for the perfect perfect-maze. In *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES).* 65–72. DOI: http://dx.doi.org/10.1109/CGames.2015.7272964

[13] P. H. Kim and R. Crawfis. 2018. Intelligent Maze Generation based on Topological Constraints. In *2018 7th International Congress on Advanced Applied Informatics.*

[14] P. H. Kim, S. Wurster, and R. Crawfis. Submitted in 2019. Maze Generation with Topological Constraints. (Submitted in 2019).

[15] Steve LaValle. *The RRT Page.* http://msl.cs.uiuc.edu/rrt/index.html.

[16] D. Maung and R. Crawfis. 2015. Applying formal picture languages to procedural content generation. In *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES).* 58–64. DOI: http://dx.doi.org/10.1109/CGames.2015.7272963

[17] Nintendo. 2017. *The Legend of Zelda: Breadth of the Wild.*

[18] Yoshio Okamoto and Ryuhei Uehara. 2009. How to make a picturesque maze. In *CCCG.*

[19] Hans Pedersen and Karan Singh. 2006. Organic Labyrinths and Mazes. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering (NPAR '06).* ACM, New York, NY, USA, 79–86. DOI: http://dx.doi.org/10.1145/1124728.1124742

[20] Dave Phillips. *Dave Phillips Mazes and Games.* https://www.davephillipsmazesandgames.com/.

[21] Walter D. Pullen. *Think Labyrinth!* http://www.astrolog.org/labyrnth.htm.

[22] Gillian Smith and Jim Whitehead. 2010. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games (PCGames '10).* ACM, New York, NY, USA, Article 4, 7 pages. DOI: http://dx.doi.org/10.1145/1814256.1814260

[23] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (September 2011), 172–186. DOI: http://dx.doi.org/10.1109/TCIAIG.2011.2148116

[24] L. Wan, X. Liu, T. Wong, and C. Leung. 2010. Evolving Mazes from Images. *IEEE Transactions on Visualization and Computer Graphics* 16, 2 (March 2010), 287–297. DOI: http://dx.doi.org/10.1109/TVCG.2009.85

[25] Jie Xu and Craig S. Kaplan. 2006. Vertex Maze Construction. *Journal of Mathematics and the Arts* (November 2006).

[26] Jie Xu and Craig S. Kaplan. 2007. Image-guided Maze Construction. *ACM Trans. Graph.* 26, 3, Article 29 (July 2007). DOI: http://dx.doi.org/10.1145/1276377.1276414