Intelligent Maze Generation

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Paul Hyunjin Kim, B.S., M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2019

Dissertation Committee:

Roger Crawfis, Advisor

Matthew Lewis

Jian Chen

Abstract

A maze is a puzzle in which a player finds a path from the starting point to the ending point. These days, maze is not only used as a puzzle but also adopted in many different fields. In the field of computer games, it can be used as a basic structure for a game level. In the field of robotics, it can be used as a platform to demonstrate the robot's learning ability. Also, in the field of architecture, its pattern can be used to decorate a building. Maze users in different fields may have different purposes to use a maze and need different properties of the maze. For example, a user may want a maze with numerous turns and branches in a robot contest, while others may want a maze with long straight passages and fewer dead-ends in building decoration. Thus, our research focuses on developing a method which can generate a maze based on the desired properties. Our research domain is perfect maze. Perfect maze is one type of a maze that has neither loops nor inaccessible areas. To generate a perfect maze satisfying given desired properties, our research applies a search-based procedural content generation (SBPCG) approach. SBPCG is one approach to procedurally generate game content like weapons and terrains via a searching mechanism. In our research domain, the perfect maze, SBPCG searches/generates perfect mazes until either a satisfactory maze is found or a termination criteria is met. Since a perfect maze has a structure of a spanning tree, spanning tree generation algorithms, such as Prims algorithm and Kruskals algorithm, can be used. This research aims to investigate whether these spanning tree generation algorithms are capable of generating the desired mazes using

SBPCG approach. Since they search mazes randomly in the space of spanning trees, mazes with some desired properties could result in infinite searching. Thus, our research also provides new method, which applies intelligent searching mechanism in SBPCG approach. In this new method, from a space of spanning tree, we pre-build several representative vectors and use them in a SBPCG approach to find the desired mazes with very high probability. Our research demonstrates that any desired mazes can be found effectively with this representation-based method. Additionally, we also provide a method which can control the solution path topology on a maze so that users can manage game quality. To demonstrate how useful our intelligent maze generation is, this thesis provides several use-cases of building actual game levels and shows that the levels can be designed effectively using our method.

I dedicate this thesis to my God who gave me the strength, wisdom and patience to complete this study, to my beloved family for their love and support.

Acknowledgments

Firstly, I would like to give sincere thanks to my God for giving me the chance of doing this Ph.D. course and upholding me with the right hand of his righteousness to complete this Ph.D. research.

Next, I would like to express my sincere gratitude to my advisor Dr. Roger Crawfis. His guidance helps me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Matthew Lewis and Dr. Jian Chen, for their insightful comments and hard questions that make me think my research deeply.

My sincere thanks also goes to my previous labmates: David Maung and Chloe Shi, who helped me to conduct this research in a good lab environment. Also, I thank to Jacob Grove for developing fancy maze design tool. Especially, I am grateful to Skylar Wurster for editing my research papers and improving the papers with beautiful figures.

I am also grateful to the university staffs in the department of computer science and engineering. In particular, I give sincere thanks to Kathryn Reeves who really helped me to get the funding to finish my Ph.D. study. Without you, I could not have imagined to continue this Ph.D. course.

Also, I thank my friends in The Ohio State University and in Korean church for their kind support and encouragement.

I would like to thank my family for providing me through moral and emotional support. And finally, last but not the least, I sincerely thank the most precious women: my beloved wife Gunyoung Kim and my beloved daughter Daniela Kim for being my everything in my life.

Vita

June 16, 1984	Born - Tennessee, USA
2009	B.S. Mechanical Engineering, College of Engineering Korea University, Seoul, South Korea
2011	M.S. Graduate School of Culture and Technology, KAIST, Daejeon, South Korea
2011-present	PhD Student, Computer Science and Engineering, The Ohio State University, Columbus, OH.

Publications

Research Publications

Paul Hyunjin Kim, Skylar Wurster, and Roger Crawfis "Maze Generation Control based on Desired Topological Properties". *To be submitted*

Paul Hyunjin Kim, Jacob Grove, Skylar Wurster, and Roger Crawfis "Design-Centric Maze Generation". *The 10th Workshop on Procedural Content Generation (PCG2019)*, San Luis Obispo, CA, Aug. 2019.

Paul Hyunjin Kim and Roger Crawfis "Intelligent Maze Generation based on Topological Constraints". *7th International Congress on Advanced Applied Informatics* , Yonago, Japan, Jul. 2018

Paul Hyunjin Kim and Roger Crawfis "The Quest for the Perfect-Perfect Maze". *Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*, Louisville, KY, Jul. 2015.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

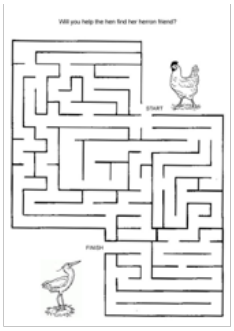
Table **Page**

List of Figures

Chapter 1: Introduction

A maze is a puzzle which challenges players with its simple goal: Find a path from an entrance to a goal on a maze. A maze puzzle consists of passages, an entrance, and a goal. By these maze components, the simple goal of a maze becomes intuitively obvious, and even children can play in a maze without a deep understanding of mazes. Despite its low barrier to entry, a maze also can be difficult, such that even adults need to spend several hours to solve a certain maze. Due to its simple goal and challenging aspects, mazes are being enjoyed as a puzzle regardless of ones age. There are various types of maze puzzles as shown in Figure 1.1. In the paper maze shown in Figure 1.1(a), a player solves the maze via a top viewpoint. In the hedge maze shown in Figure 1.1(b), which is a life-size maze, a player walks through passages via a first-person ground level viewpoint to solve the maze. There is also a maze printed on a silicon phone case, as shown in Figure 1.1(c), which provides both amusement and protection for phone users.

A maze is a simple puzzle, but it is also used as a tool in different fields, as represented in Figure 1.2. In computer games, such as Pac-Man and The Legend of Zelda, a maze is used as a basis for levels, as shown in Figure 1.2(a). In the robot contest Micromouse [3], robot mice compete for their path of learning ability, and a maze is used as a platform for the contest, as represented in Figure 1.2(b). The micro robot who records the fastest time to arrive at a designated point in a maze becomes the winner in the contest. In the field of



(a) Paper Maze

(b) Hedge Maze

(c) Maze Phone Case

Figure 1.1: Maze puzzles in the world.

architecture, there are designs which use a maze to decorate buildings, as shown in Figure 1.2(c). As we can see in these examples, a maze has the possibility to be used differently in various domains.

Since a maze can be used for different purposes, when users create a maze, they may want to incorporate different properties on a maze based on its purpose. For example, a user may want a maze with numerous turns and branches in a robot contest, while others may want a maze with long straight passages and fewer dead-ends in a building decoration.

Our research problem is that: in different fields, when users have different desired properties of a maze, how can we generate a maze that best fits their own purpose? Our research domain is a perfect maze. The maze has a spanning tree structure which has no loops and no inaccessible areas. Our research focuses on a maze without a loop, but if loops are wanted, we can simply adding them on the maze via post-processing. In our research, the desired properties of a perfect maze include topological properties of the perfect maze such as the number of branches, the amount of homogeneity, and the length of a solution path in the maze.

Figure 1.2: Different fields using a maze.

One way to create a maze with the desired properties is to manually construct the desired maze. However, it could be very difficult for a novice to manually draw a maze. Kaplan [21] stated that when a professional maze designer creates a maze, drawing the walls of a maze may be based purely on intuition which includes an implicit understanding about how passages of a maze should be connected. Thus, as talked in [11], substantial effort and practice is required to be familiar with creating a maze manually.

Instead, we can apply a computer-aided approach. There have been studies in which the shape of a maze is determined based on an input parameter, a 2D graphic, using a computer. Xu and Kaplan [38] developed a method to generate a maze structure embedded into a given 2D image. In [30], a labyrinth, which is a maze with a single long windy passage, was generated by evolving curves based on given texture maps. In [28], a maze was generated based on an input raster image, so that filling up a solution path of the maze reveals the input image.

Although the above studies are visually appealing, these studies do not ensure that the maze has desired topological properties, such as desired number of dead-ends on a maze. In our research, we introduce a search-based procedural content generation (SBPCG) approach to generate a maze with desired topological properties. SBPCG is one approach to procedurally generate game content, such as dungeons, weapons, and terrains. SBPCG applies a searching mechanism to find satisfactory contents in a search space. In our domain of a perfect maze, SBPCG approach generates a perfect maze and checks whether the generated result satisfies the desired properties. The searching mechanism continues until the ideal maze is found. Or, it can continue until some amount of mazes are generated, and the satisfactory one among the generated mazes is returned.

When we generate mazes in the searching process of the SBPCG approach, we can use several perfect maze generation algorithms described in [15], in which most of them are based on spanning tree generation algorithms. The algorithms randomly generate mazes in a search space.

In our research, we investigate if these random algorithms are capable to generate desired mazes using SBPCG approach. Then, we develop a method that utilizes existing maze generation algorithms to generate desired mazes. However, since each algorithm tends to generate mazes with specific properties, mazes with some desired properties could result in infinite searching. Thus, our research introduces new intelligent searching method. This method pre-constructs several representations from enumerated set of spanning trees and use them in SBPCG approach to find any desired mazes effectively. We also provides a way to control the solution path topology on a maze. To demonstrate the usefulness of our intelligent maze generation, we also provide several use-cases of building actual game levels and show how we can use our method to design the levels.

Chapter 2: Related Works

There are many methods focusing on solving a maze, but this paper focuses on procedurally generating a maze. In the real world, many mazes are created manually. We can easily see lots of maze books, where mazes are drawn by hand. In [11], a maze designer Christopher Berg shows his astonishing maze works created manually on paper. In [19] and [31], 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.

But now there are many computer algorithms to draw a maze automatically. In [32] and [15], a comprehensive list of maze generation algorithms is provided. Most of these are based on spanning tree algorithms. As explained in [32] and [15], classic spanning tree algorithms, such as Prim's algorithm, are used as the maze generation algorithms. In [14], it animates the maze generation process of the algorithms so that we can see the steps of the algorithms easily. In the RRT Page [25], rapidly-exploring random trees are introduced, and mazes that created using the trees are shown.

Research has been conducted on developing the aesthetic aspects of a maze. Xu et al. [38] constructed a maze with obfuscated structure by adopting a vortex shape on it. Also, Pedersen et al. [30] created organic labyrinths, which resemble input images, by evolving

simple curves. In [39] and [36], a maze was generated based on a given image. Consequently, the input image is filled with paths of a maze. In [17], paths of a maze were decorated with textures designed by artists. In [28], 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. ([9], [10]) developed an evolutionary algorithm-based system, in which a maze-like level is evolved with dynamic programming to control the resulting structure. In [9], mazes are evolved to have a specified level of difficulty, in which the difficulty is measured by the number of steps to solve a maze. [10], Ashlock et al. introduce multiple maze representations and show how the generator has different tendencies over generated maze structures based on representation. Bosch et al. [12] proposed a method that utilizes integer programs to generate a labyrinth with path-segment tiles. In integer programs, we can input some constraints to force the labyrinth to be as symmetric as possible. Kaplan [21] did some research regarding a reconfigurable maze, where rotating maze parts yields a maze with another topology. In [26], Maung et al. used array grammar rules to define a path structure, and a maze was generated based on the rules. Chad et al. [7] evolved cellular automata rules using genetic algorithms to create mazes with longer paths. Kim et al. [22] 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 of the desired properties effectively.

Chapter 3: Research Domain

Our research domain is a perfect maze on a regular grid. A perfect maze is one type of a maze, whose structure has characteristics of a spanning tree. Spanning tree is a graph where all nodes of the graph are connected by its edges without any loops. An example of a perfect maze on a regular grid is shown in Figure 3.1. In this chapter, we provide detailed information about our research domain.

3.1 Regular Grid

In this section, we define a regular grid which is a platform of our research domain, perfect maze. A regular grid is a grid which consists of rectangle cells. An MxN regular grid denotes a grid with M columns and N rows of rectangle cells. An example of a 4x4 regular grid, which has a total 16 cells, is shown in Figure 3.2.

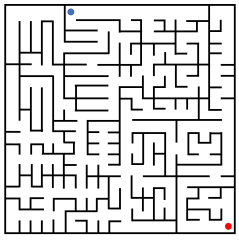


Figure 3.1: An example of a perfect maze on a regular grid with the starting point (top) and the ending point (bottom).

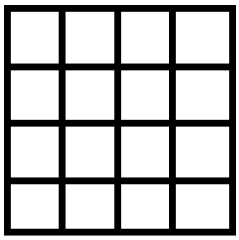


Figure 3.2: Representation of a 4x4 regular grid. Each small rectangle denotes each cell.

3.2 Spanning Tree on Regular Grid

In this section, to illustrate a structure of perfect maze on a regular grid, since the structure has properties of a spanning tree, we explain a spanning tree on a regular grid. In [5], a spanning tree is defined as a subgraph of graph G that connects all nodes of G by its edges without any loops. In an MxN regular grid of Figure 3.2, spanning tree is a subgraph of MxN grid graph. First, we explain what a grid graph is. As shown in Figure 3.3(a), a grid graph is a graph on a regular grid, whose nodes are at center of cells of grid, and whose edges are placed between its nodes either vertically or horizontally. Please note that when we mention nodes of a grid or edges of a grid, it denotes nodes of a grid graph on the grid or edges of a grid graph on the grid, respectively. As shown in Figure 3.3(b), a spanning tree on a regular grid has subset of edges of a grid graph on the grid, where edges of the spanning tree connect all nodes of the grid graph without any loops. Since there is no loop on a spanning tree, a unique path exists from any node to any other node of a spanning tree. As explained in [18], a spanning tree has $|V| - 1$ edges, where $|V|$ denotes the number of nodes of the spanning tree. Thus, in a MxN grid, since $|V| = M \times N$, the number of edges of a spanning tree is $M \times N - 1$.

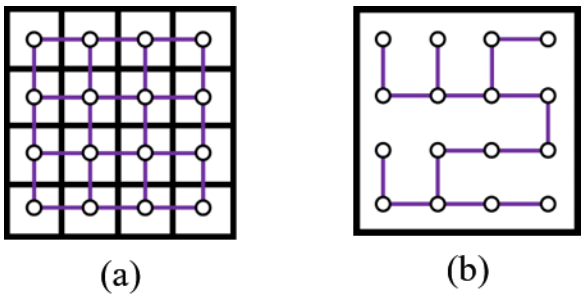


Figure 3.3: (a) Representation of a grid graph on 4x4 regular grid. Each node of the grid graph is denoted by circle, and each edge of the grid graph is denoted by purple edge. (b) An example of a spanning tree in a 4x4 grid. It consists of subset of edges of the grid graph in (a).

3.3 Perfect Maze with Spanning Tree

Perfect maze is a maze, whose structure has spanning tree properties. Like a spanning tree, a perfect maze does not have a loop, and all nodes on the perfect maze are reachable, which means there is no inaccessible node on the maze. Thus, a player can go from one node to any other node in a perfect maze, and there is only one route between these nodes.

On a regular grid, a perfect maze can be generated from a spanning tree. When a spanning tree is generated on a regular grid, the perfect maze is created by choosing any two nodes of the spanning tree for the start point and the end point of the maze as represented in Figure 3.4. Edges of the spanning tree become paths of the perfect maze. In an MxN grid, the total number of all possible choices for a pair of starting and ending points of the maze is $\binom{M\times N}{2}$. Hence, we can produce $\binom{M\times N}{2}$ perfect mazes for each spanning tree.

3.4 Maze Components

In this section, we introduce several components that we can have on a perfect maze. Here, some descriptions are given from the viewpoint of a player playing a maze puzzle.

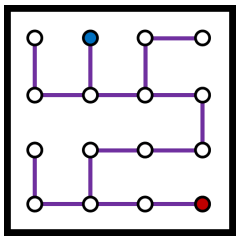


Figure 3.4: Perfect maze generated from the spanning tree in Figure 3b with the start (blue point) and the end (red point).

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 a player needs to arrive to finish a 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: The paths on a perfect maze, except 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. This component can delude a player that he/she follows the solution path toward the exit.
    - 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.

When we play a maze, if the location of the end point is shown for a player, forward dead-end and backward dead-end follows the above definitions. However, if the location of the end point is visually occluded for a player, such as a 3D maze with first-person view, we define a forward dead-end as a dead-end which has forward direction from current location and a backward dead-end as a dead-end which is neither a forward dead-end nor an alcove.

4. Maze Cells: A maze cell is a unit of a perfect maze, and there can be five types of maze cell based on a shape of paths on its center. Each type of maze cell is illustrated in Figure 3.5, 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 cells as shown in Figure 3.5(a).
    - Straight: Two paths on the cell center have the same direction. There are two types of straight cell: 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 paths 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 3.5(c). In a perfect maze, when a player enters a T-junction, the player needs to decide which path to take.
    - Cross-Junction: It has four paths out of the cell center. The player has three choices of direction to decide from in a perfect maze.

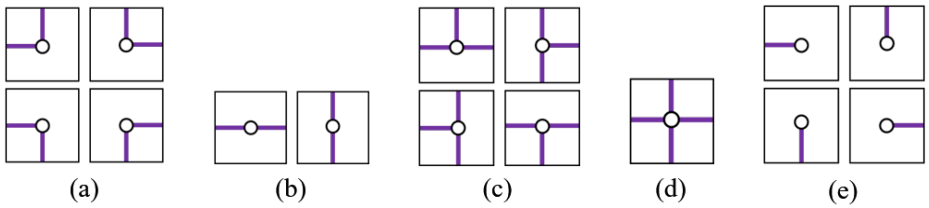


Figure 3.5: Representation of types of maze cell on a regular grid ((a) Turn, (b) Straight, (c) T-Junction, (d) Cross-Junction, and (e) Terminal). The black square represents a cell, and the purples line represents an edge. Also, the black circle represents a node.

- 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 3.5(e).

As we can see in this chapter, perfect maze and spanning tree have duality on a grid graph. Thus, we can see perfect maze problem as spanning tree problem. Thus, we will use both terminologies interchangeably in this thesis. The difference is that a maze has start and end cells.

3.5 Game Level Creation with Perfect Maze Topology

When a perfect maze is created with a spanning tree on a regular grid, to present the perfect maze more like a game level as shown in Figure 3.1, we can place walls based on topology of the perfect maze. When neighboring nodes are not connected by an edge on a perfect maze, a wall is placed between the nodes on a grid. For example, as shown in Figure 3.6, since node A is not connected to node B by an edge, a wall is placed between nodes A and B on a game level. But, since node A and C is connected directly, a wall is not placed between nodes A and C on a game level.

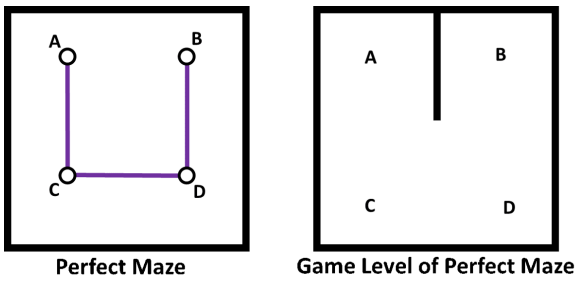


Figure 3.6: An example of a perfect maze, where cells A and B are connected through the other nodes C and D in a 2x2 grid, and its game level representation.

Figure 3.7 illustrates creating a game level from a given perfect maze in 3x3 grid. When a perfect maze, which is a spanning tree with the start and the end, is given as shown in Figure 3.7(a), walls are placed between nodes, in which they are connected through other nodes on a perfect maze, as shown in Figure 3.7(b). When all possible walls are placed on a grid, a game level is created by erasing all edges and nodes of the perfect maze except the starting point and the ending point as shown in Figure 3.7(c).

Likewise, instead of adding walls, by carving passages, we can create an actual game level from a perfect maze as shown in Figure 3.8. In this case, first we place walls between all nodes as shown in Figure 3.8(a) and then remove a wall (carve a passage) between nodes that are connected by an edge as represented in Figure 3.8(b). After we remove all walls between connected nodes on a grid, we can obtain maze-like game level as described in Figure 3.8(c).

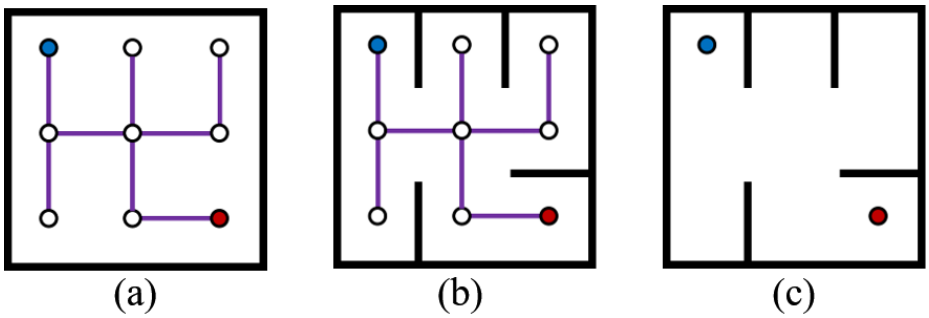


Figure 3.7: (a) a perfect maze built based on a spanning tree in a 3x3 grid. (b) Walls are placed on the perfect maze. (c) Game level generated from the perfect maze.

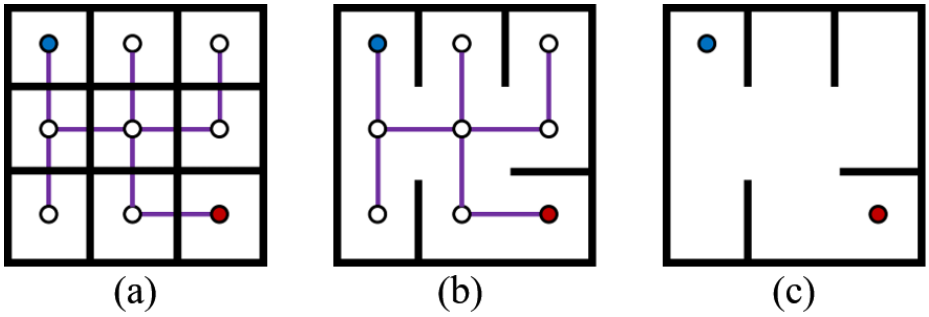


Figure 3.8: (a) Walls are placed between all nodes of a perfect maze on a 3x3 grid. (b) Walls are removed between connected nodes on the perfect maze. (c) Game level generated from the perfect maze.

Chapter 4: Defining Desired Properties

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

4.1 Basic Properties

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

Some simple metrics just convert the number of cells with each topology. These can either be explicit or represented as a percentage of the overall of cells in the 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 on a maze.
- #Straights: Denotes the number or percentage of straight cells on a maze.
- #T-Junctions: Denotes the number or percentage of T-junction cells on a maze.
- #Cross-Junctions: Denotes the number or percentage of cross-junction cells on a maze.
- #Terminals: Denotes the number or percentage of terminal cells on a maze.

Solution Path

- Starting Cell and Ending Cell
- #TurnsSP: The number or percentage of turn cells on the solution path.
- #StraightsSP: The number or percentage of straight cells on the solution path.
- #T-JunctionsSP: The number or percentage of T-junction cells on the solution path.
- #Cross-JunctionsSP: The number or percentage of cross-junction cells on the solution path.
- SolutionPathLength(SPLength): The solution path length, which is measured by the number of edges on the solution path. Easier semantics also exist such as short, long, and very long.
- Max#ContiguousStraightCells: Denotes the maximum number or percentage of contiguous straight cells where no turn cell exists between the straight cells on the solution path.
- #DETrees: The number of dead-end trees branched from the solution path. This is equivalent to $\#T - JunctionsSP + \#Cross - JunctionsSP * 2$.

Dead-End Tree

- DESize: The total number or percentage of cells on all dead-end trees. It excludes the cell attached on the solution path.
- #TurnsDE: The number or percentage of turn cells on dead-end trees.
- #StraightsDE: The number or percentage of straight cells on dead-end trees.

- #T-JunctionsDE: The number or percentage of T-junction cells on dead-end trees.
- #Cross-JunctionsDE: The number or percentage of cross-junction cells on dead-end trees.
- #TerminalsDE: The number or percentage of terminal cells on dead-end trees.

4.2 Higher-Level Properties

Additionally, we can define higher-level topological properties on a maze, which are described in [32] and [23]. 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 a maze. A high river factor indicates that there are a few long dead-ends on a maze.
- 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 texture. High homogeneity yields a maze with homogeneous textures, but low homogeneity gives a maze with diverse textures.

- Hidden Factor: Indicates how hard it is to recognize the shape of paths from any place while playing a maze in a first-person view. 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.

These high-level properties contain some design-centric concept, and some of them can be defined using the above basic metrics (#Turns, #Straights, #T-Junctions, #Cross-Junctions, #Terminals). For example, to have a higher run, we may give a large value of #Straights. Also, to have a high river factor, we may desire small values of #T-Junctions and #Terminals. However, a property like a hidden factor has some cognitive term, and it would be hard to define this kind of property quantitatively with the basic metrics, and is one of the future works to explore.

Chapter 5: Maze Generation Algorithms

In this chapter, we provide existing maze generation algorithms. Since a perfect maze and a spanning tree have duality as mentioned in Chapter 3, we can use existing spanning tree algorithms for maze generation. Thus, we introduce existing spanning tree algorithms and provide an overview of how each algorithms processes. For more detailed explanation about algorithms to generate mazes, please refer to [32] and [15]. Note that when an algorithm has separate resources for reference, I leave the citations in the corresponding section.

5.1 Spanning Tree Algorithms

5.1.1 Recursive backtracker

The process starts at any node in a grid and moves to one of the neighboring nodes that has not been visited yet. In a grid, each node has at most four neighboring nodes that are connected by edges. After the process moves to the current node, it recursively moves to an unvisited neighboring node until it visits a node that has been visited previously. When the node u does not have unvisited neighbors, it backtracks to the node v where the process visited right before the node u, and the edge between u and v becomes an edge of a spanning tree. Then, the process recursively moves to other unvisited neighboring nodes from the node v until it visits a node that has been visited before. The process continues until all the nodes in a graph are visited. When all the nodes are visited, the process backtracks to

the starting node and stops. During backtracking to the starting node, edges on the path can be edges of a spanning tree (See Figure 5.1). Once the process stops at the starting node, a spanning tree has been generated.

The progress of recursive backtracker in a 4x4 grid is shown in Figure 5.1. In Figure 5.1, the red node denotes the current location of the process. The blue nodes indicate neighboring unvisited nodes of the current node (red node). The yellow nodes mean visited nodes. The purple edge shows the path in which the process passed and is going to backtrack later, and the black edge means an edge of a spanning tree. When the red node does not have blue neighbors in Figure 5.1(f), the process backtracks to the prior node in Figure 5.1(g), and the purple edge in which the process backtracked becomes the black edge in Figure 5.1(h). Then, the red node moves to other blue neighbor, as shown in Figure 5.1(h), and recursively moves to unvisited nodes as shown in Figure 5.1(i). When all nodes are visited in Figure 5.1(j), the process backtracks to the starting point, and we obtain a spanning tree in Figure 1n.

5.1.2 Prim's algorithm

Initially, we have an empty edge list EL, where edges are temporarily stored and removed. This algorithm starts at any node. When a node is chosen to start, the adjacent edges connected to the current node are added to EL. Then, the process selects an edge in EL randomly and checks if the edge has an unvisited adjacent node. The selected edge is removed from EL, and if an adjacent node of the selected edge has not been visited yet, the edge becomes an edge of a spanning tree. The process visits the adjacent node, and the neighboring edges of this new node that have not been selected previously are added to EL. If an adjacent node of the selected edge has already been visited, the edge will form a loop with edges of a spanning tree, so the edge does not become an edge of a spanning

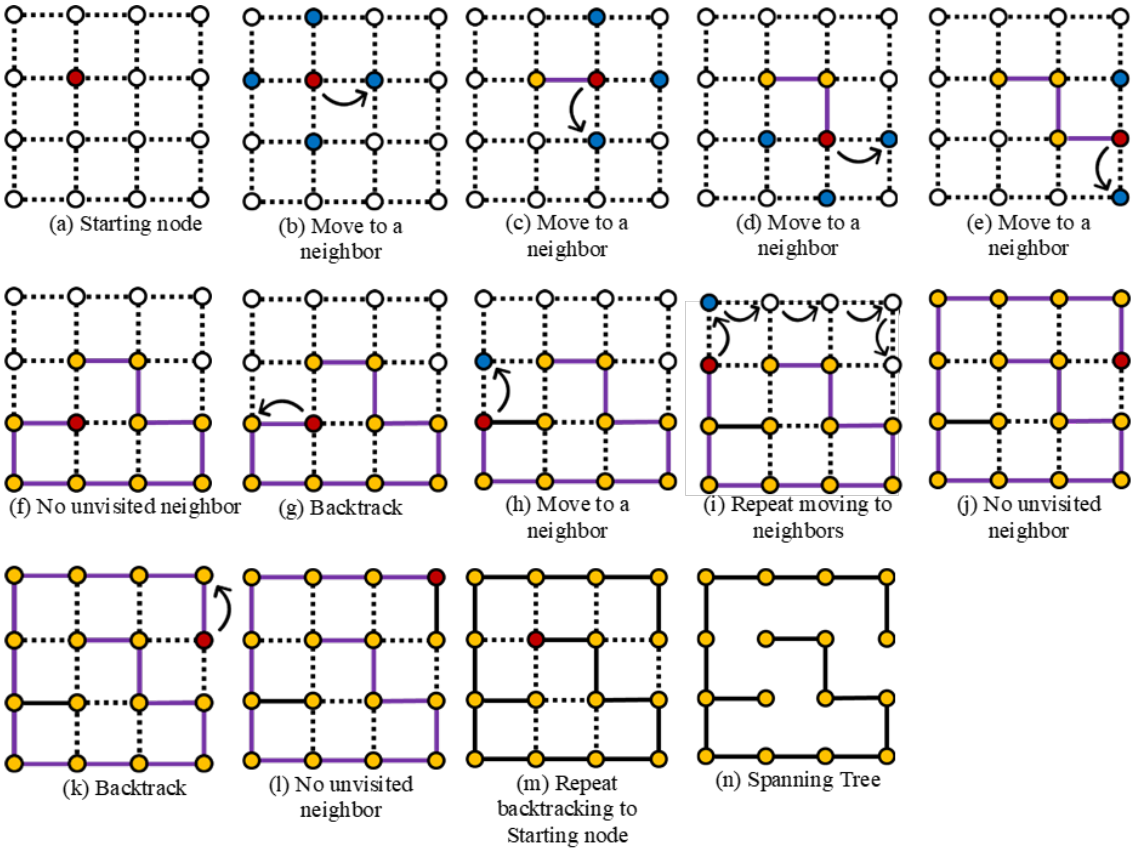


Figure 5.1: The progress of recursive backtracker.

tree. Then, another edge is selected randomly and removed from EL again, and it checks whether a neighboring node of the selected edge has been visited yet or not. The process repeats until all nodes are visited. The algorithm ends with a spanning tree.

The progress of Prims algorithm in a 4x4 grid is illustrated over Figure 5.2 and Figure 5.3. In Figures 5.2 and 5.3, the red node is the current location of the process, and the yellow nodes are the nodes that the process visited. The blue edges denote edges in EL. The purple edge is the edge selected by the process among edges of EL. The black edge represents an edge of a spanning tree. The small number on each edge of a grid means an index of each edge, and the edge index is added to EL. As shown in Figure 5.2(b), the blue adjacent edges of the red node are added to EL. Then, as shown in Figure 5.2(c), the process selects the edge denoted by purple color among blue edges in EL, and the purple edge is removed from EL. Since the adjacent node of the purple edge is unvisited node, the red node moves to the adjacent node, and the purple edge becomes the black edge as shown in Figure 5.2(d). As shown in Figures 5.2(l) and 5.2(m), when the neighboring node of the red node on the purple edge is yellow node (visited node), since the purple edge will form a loop with black edges, the purple edge does not become a black edge. The process stops when all nodes are visited as shown in Figure 5.3(d), and, as a result, a spanning tree in Figure 5.3(e) is obtained.

5.1.3 Kruskal's algorithm

At the beginning, each node is contained in its own set. Assume that we have a group of all edges of a grid. An edge is randomly selected and removed in the group. If both adjacent nodes of the selected edge are in different sets, the edge becomes an edge of a spanning tree, and these sets are merged into the same set. If both adjacent nodes of the selected edge are in same sets, because the edge is going to form a loop, the edge is just

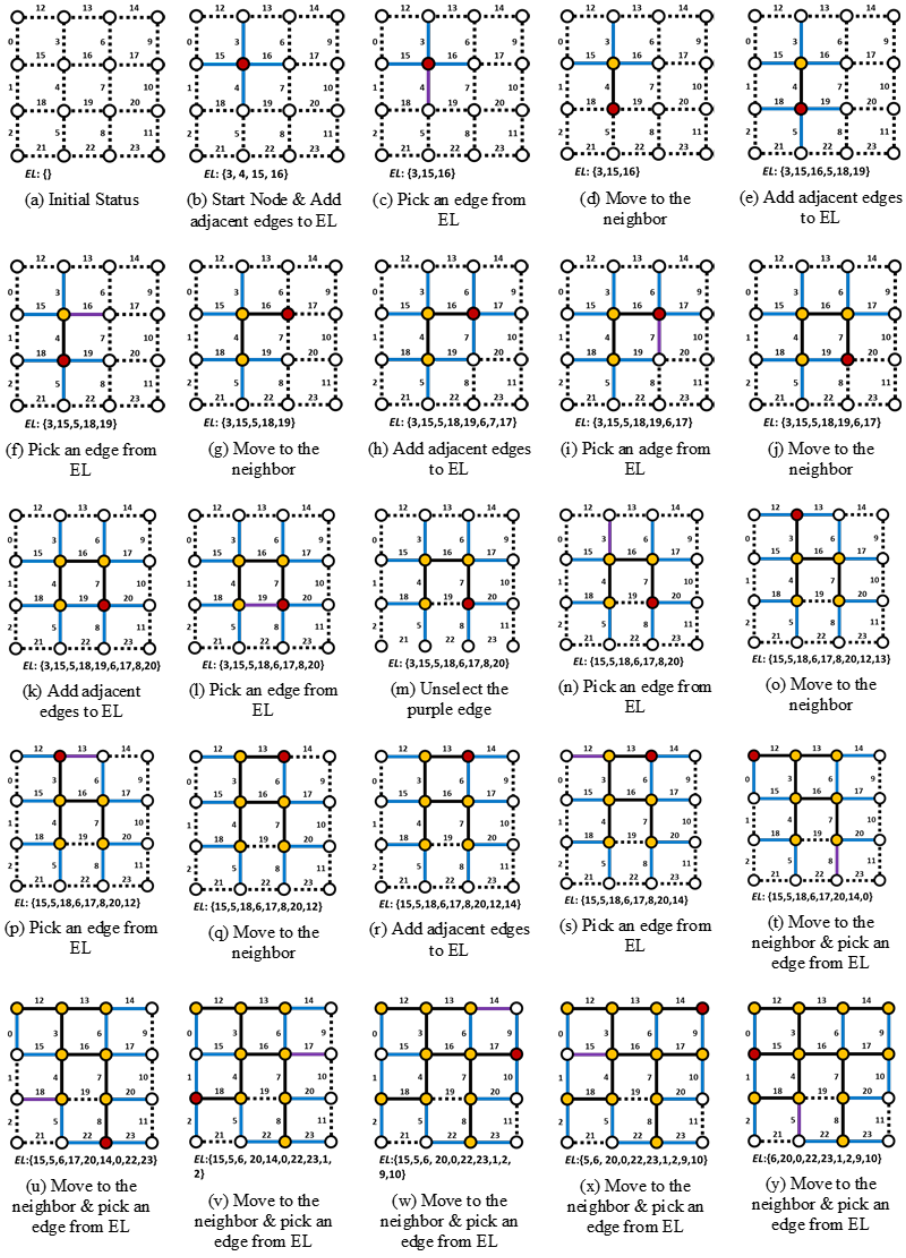
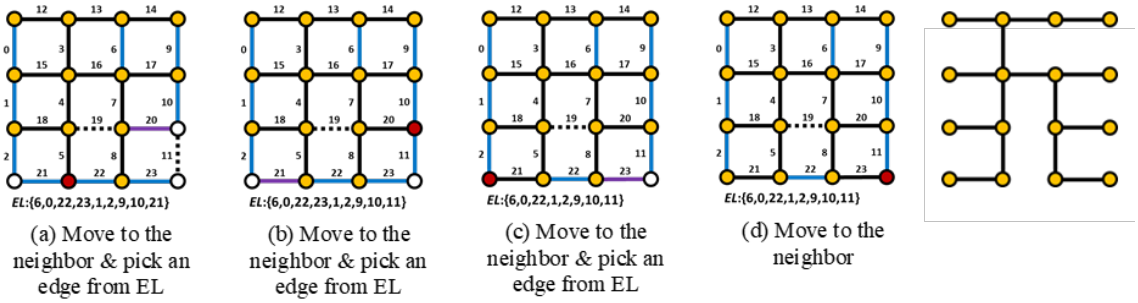


Figure 5.2: The progress of Prim's algorithm.



(a) Move to the neighbor & pick an edge from EL

(b) Move to the neighbor & pick an edge from EL

(c) Move to the neighbor & pick an edge from EL

(d) Move to the neighbor

Figure 5.3: The progress of Prim's algorithm (continued).

removed from the edges' group. This algorithm stops once all nodes are in the same set. Consequently, the process yields a spanning tree.

The progress of Kruskal's algorithm is shown in Figure 5.4. In Figure 5.4(a), each node has each own color which means that each node belongs to each own set. If nodes have the same color, it means that the nodes belong to the same set. The purple edge denotes randomly selected edge, and the black edge represents an edge of a spanning tree. Please note that, as shown in Figure 5.4(b) and 5.4(c), when adjacent nodes of the purple edge have different colors, the purple edge becomes the black edge, and both nodes become to have the same color. In Figures 5.4(f) and 5.4(g), when the purple edge is going to form a loop with black edges, the purple edge does not become a black edge. The process ends with a spanning tree in Figure 5.4(s).

5.1.4 Binary Tree algorithm

Initially, the process begins at any node on a grid. Then, the process chooses one of two adjacent edges which are perpendicular to each other, and the selected edge becomes an edge of a spanning tree. There are four cases where two edges are perpendicular in a grid,

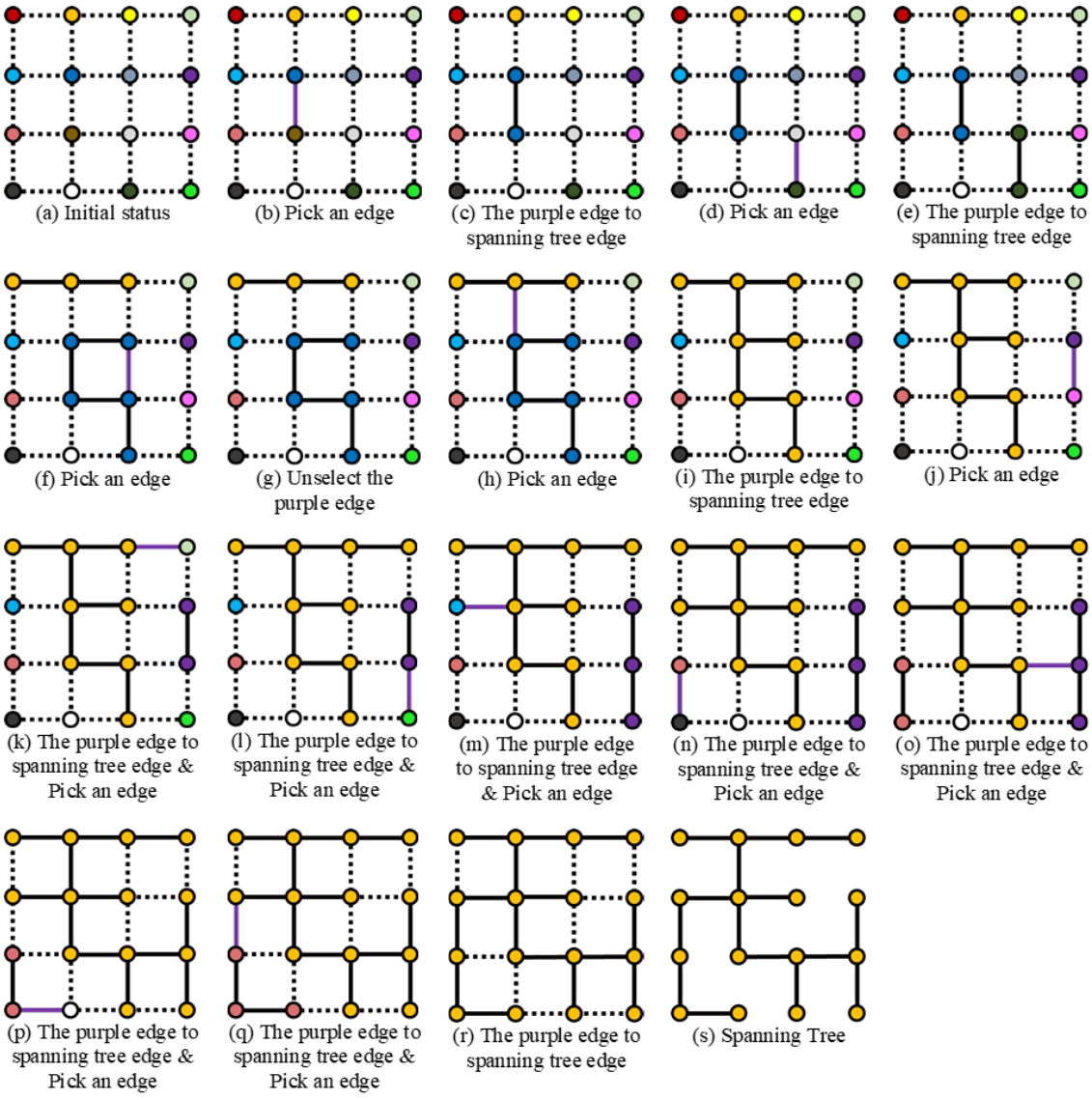


Figure 5.4: The progress of Kruskal's algorithm.

top edge and right edge, top edge and left edge, bottom edge and right edge, and bottom edge and left edge. In this explanation, we choose the edge between top edge and right edge in edge selection process. After an edge is selected between top edge and right edge of the current node, the process visits any unvisited node next. Then, the edge is selected between top edge and right edge of the visited node again. After the edge is selected, the process does not automatically move to the adjacent node of the selected edge. Thus, the adjacent node of the selected edge is not marked as visited. When all nodes in a grid are visited, the process stops and yields a spanning tree.

The progress of the binary tree algorithm is shown in Figure 5.5. The red node denotes the current location of the process. The yellow node denotes the node which has been visited by the process. The blue edges represent adjacent edges of the red node considered in the edge selection process. The black edge means an edge of a spanning tree. In Figure 5.5, the top edge and right edge of the red node become blue edges. As shown in Figures 5.5(g), 5.5(h), 5.5(i), 5.5(j), 5.5(k), and 5.5(l), when the process visits the node at the boundary of a grid, it is forced to select an edge in a single direction. In Figures 5.5(g), 5.5(h), and 5.5(i), only the top edge becomes a blue edge and to be selected as an edge of the spanning tree because there is no right edge of the red node. Also, in Figures 5.5(j), 5.5(k), and 5.5(l), only the right edge becomes a blue edge and to be selected as an edge of the spanning tree because there is no top edge of the red node. If the red node does not have both top and right edges as represented in Figure 5.5(r), the process does not select an edge. An example of a spanning tree generated by this algorithm is shown in Figure 5.5(u).

5.1.5 Sidewinder algorithm

The algorithm works by rows, and the process starts either from the top-most row or the bottom-most row. Assume that the process starts from the top-most row in this explanation.

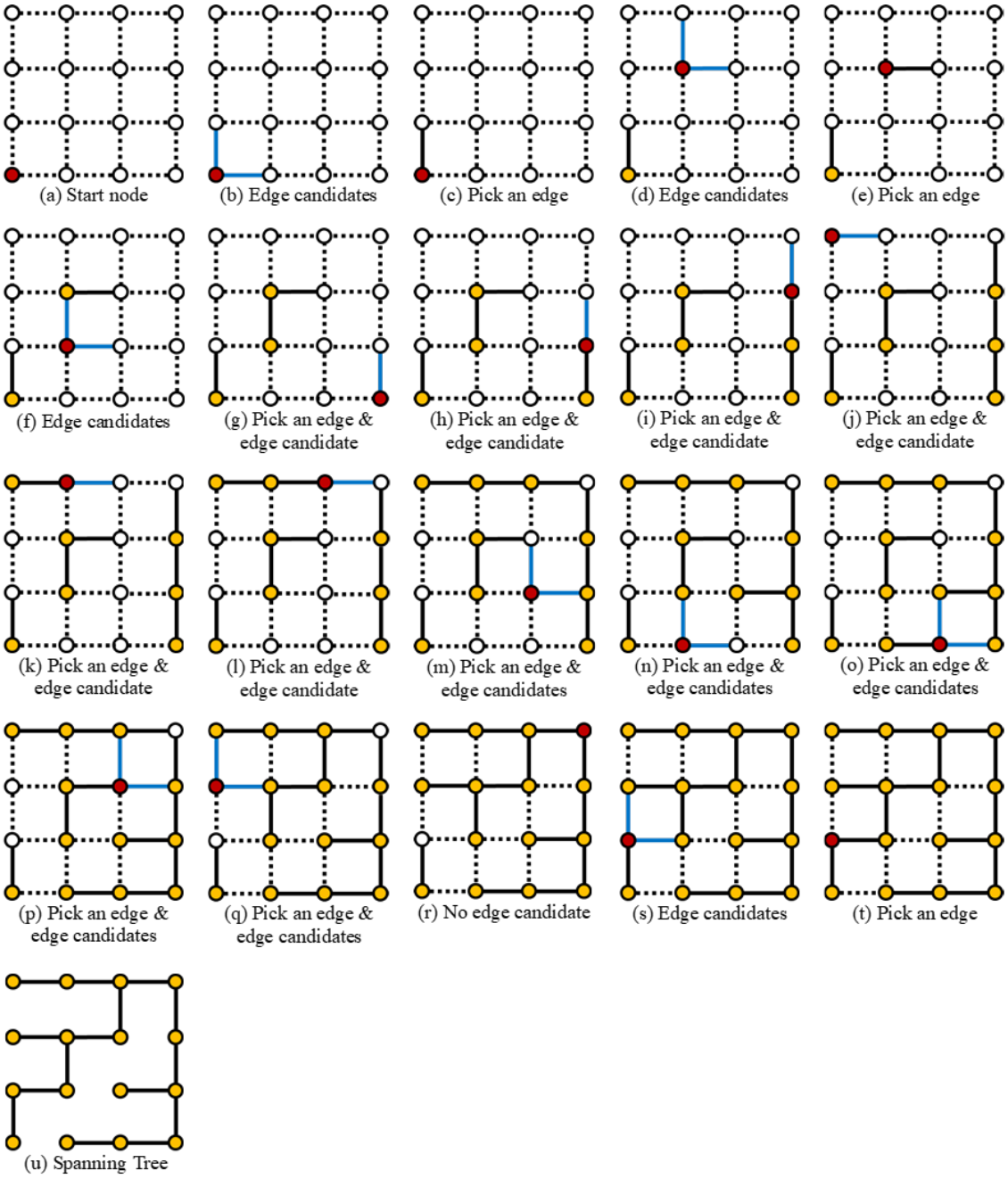


Figure 5.5: The progress of the binary tree algorithm.

In the sidewinder algorithm, in a row of a grid, the process begins with empty node list NL. The process starts from the left-most node of the row, and the node is added to NL. The process decides whether it moves to the right neighboring node or not. If the process decides to move from the current node u to the right node v, the edge between u and v becomes an edge of a spanning tree, and the node v is added to NL. If the process decides not to move, a node is selected randomly from NL, and the edge between the selected node and its top-neighboring node becomes an edge of a spanning tree. Then, NL becomes empty, and the process starts again from the node which is right of previously stopped node. When the process arrives at the right-most node of the row, because the node does not have a right neighbor, the process is forced to decide not to move. Then, any node is selected from NL, and the edge between the selected node and its top neighboring node becomes an edge of the spanning tree. After the process completes its work in the row, it moves to the next row and repeats its work with an empty NL.

Although the above strategy is applied to each row in a grid, it is applied differently to the top-most row. Since all nodes on the top-most row do not have top-adjacent edges, there is nothing we can do when the process decides not to move. Thus, when the process works on the top-most row, the process is forced to move right. Then, all edges on the top-most row become edges of a spanning tree. When the process arrives at the right-most node of the top row, because it does not have both top neighbor and right neighbor, the process does nothing, and NL is empty. The process ends yielding a spanning tree when it finishes to work on all rows.

The progress of the sidewinder algorithm is illustrated in Figure 5.6. The red node means the current position of the process, and the yellow node means a visited node. The black edge means an edge of a spanning tree. The small number on each node denotes its

node index. The arrow with red X mark denotes that the process decides not to move to the right node. Please note that, in Figure 5.6(a), edges and nodes in the top-most row are already black edges and yellow nodes, respectively, because the process is forced to move to the right neighbors on the top-most row as we explained above. In Figure 5.6(d), the process decides not to move from the node 7 to the node 8. Then, as shown in Figure 5.6(e), the node 6 is selected from the NL, and the edge between the nodes 2 and 6 becomes an edge of a spanning tree. In Figure 5.6(f), the red node is at the right-most of the second row, the node 8. Since there is only one node, the node 8, in NL, the edge between nodes 8 and 4 becomes an edge of a spanning tree. Figure 5.6(t) shows an example of a spanning tree using the sidewinder algorithm. As shown in Figure 5.6(t), the sidewinder algorithm has obvious sampling bias. It always generates spanning trees, in which all edges of the top most row are edges of the spanning tree. Other spanning trees will not be generated with the sidewinder algorithm.

5.1.6 Hunt-and-Kill algorithm

The process starts from the 'kill' mode. It begins at any node u in a grid. Then, the process moves to one of the unvisited neighboring nodes, node v, randomly. The edge between nodes u and v becomes an edge of the spanning tree. The process sequentially moves to unvisited neighbors until it reaches a node where there is no unvisited neighbor. This is the kill mode. The process, then, enters the hunt mode. In the hunt mode, the process scans a grid to find an unvisited node, w, which is neighboring to one of the visited nodes, the node x. The process visits the node w, and the edge between the nodes w and x becomes an edge of the spanning tree. Then, it enters the 'kill' mode and moves to one of its unvisited neighbors again. The process stops when all nodes in a grid are visited.

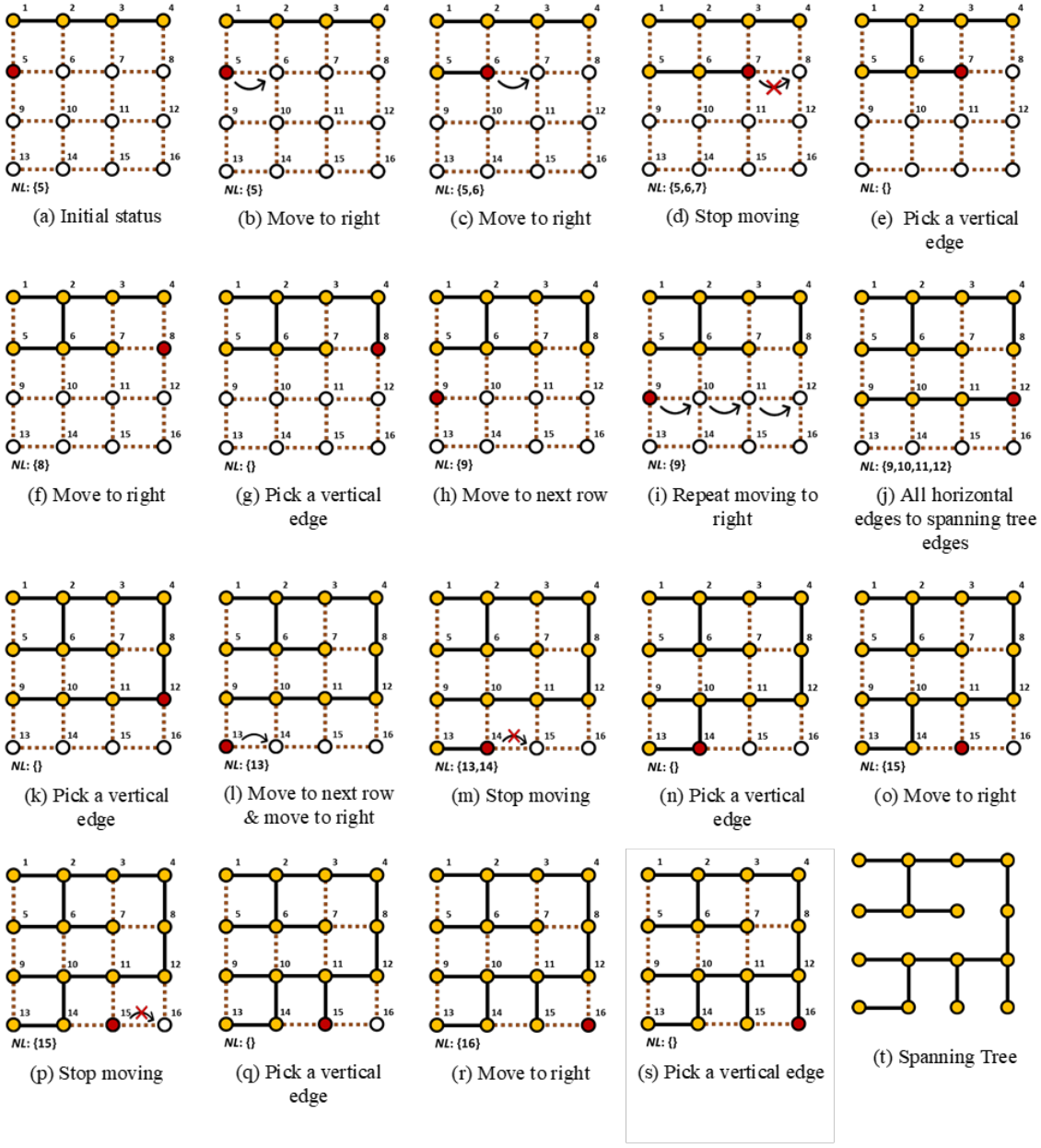


Figure 5.6: The progress of the sidewinder algorithm.

The progress of the Hunt-and-Kill algorithm is illustrated in Figure 5.7. In Figure 5.7, the red node denotes the current position of the process. The blue nodes are candidates that the process will move to next in the 'kill' mode, and the yellow node means visited node. The black edge denotes an edge of a spanning tree. When the red node does not have blue nodes (i.e., unvisited neighbors) in the 'kill' mode as shown in Figure 5.7(h), the process scans the grid and finds another unvisited node which is neighboring to one of yellow nodes, visited nodes, in the 'hunt' mode as shown in Figure 5.7(i). Then, as shown in Figure 5.7(j), the edge between the red node and its adjacent yellow node becomes a black edge, and, as shown in Figure 5.7(k), the process restarts the 'kill' mode to move to one of blue nodes from the red node. As shown in Figures 5.7(m) and 5.7(n), when the red node does not have blue nodes, the red node moves to the last unvisited node. As shown in Figure 5.7(n), the red node has two adjacent yellow nodes. In Figure 5.7(o), one of the neighboring yellow nodes (the left node of the red node) is selected randomly, and the edge between the red node and the selected yellow node becomes a black edge. An example of a spanning tree generated by the Hunt-and-Kill algorithm is shown in Figure 5.7(p).

5.1.7 Growing tree algorithm

This algorithm starts with an empty node list NL. The process starts at any node u in a grid, and the starting node u is added to NL. Then, the process selects any node from NL and visits that node. Since there is currently only one node u in NL, the node u is selected. After the node u is selected from NL, the process visits one of its unvisited neighbors, the node v, and the node v is added to NL. The edge between two nodes u and v becomes an edge of the spanning tree. Then, any node is selected from NL again, and the process visits one of its unvisited neighbors, the node w. The node w is also added to the NL. When a node is selected from NL, if the node does not have unvisited neighbors, the node is just

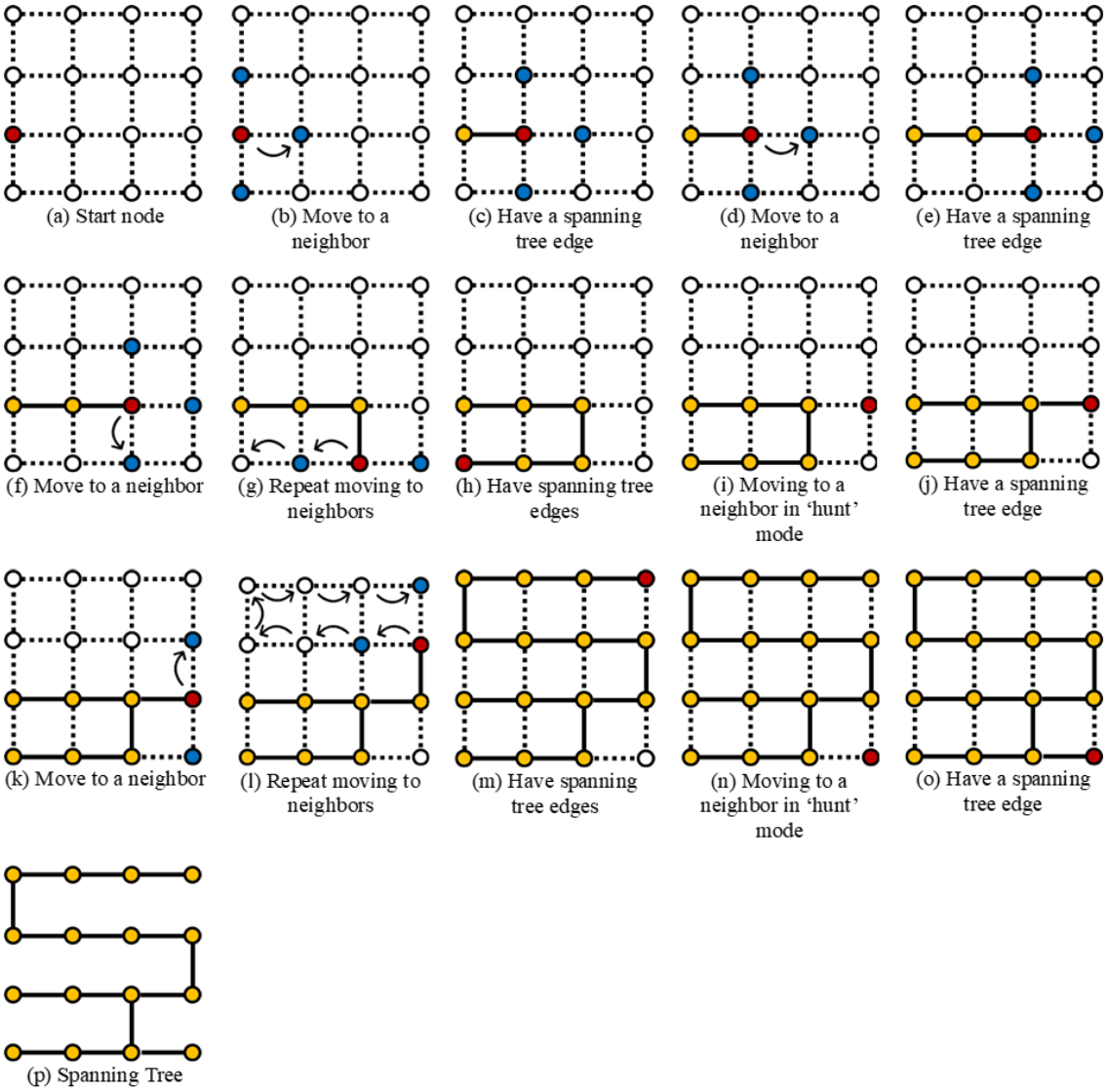


Figure 5.7: The progress of the Hunt-and-Kill algorithm.

removed from NL, and the process does not do anything. When NL becomes empty or all nodes are visited, the process stops with a spanning tree.

One characteristic of this algorithm is that we can apply different behaviors when the process selects a node from the NL. The process can select the most recently added node; it can select a node at random; or it can do some other behavior. If we decide to select the most recently added node from the NL, the process works like the recursive backtracker. The process recursively moves to unvisited neighbors by selecting recently added nodes in the NL until it visits a node that already has been visited. When it meets a visited node, since the node does not have unvisited neighbors, the node is removed from NL, and the process selects the next recently added node. Thus, the process backtracks to the prior node like the recursive backtracker. When we decide to choose a node at random from the NL, the process operated similar to Prims algorithm. Since unvisited nodes which are adjacent to visited nodes are added to NL, when the process selects a node at random from the NL, it selects and visits any unvisited neighboring node like Prims algorithm. We can mix two behaviors in the selection process and ping-pong between them. The process chooses the most recently added node the first time. Then, it chooses another node randomly the next time.

The progress of the growing tree algorithm is shown in Figure 5.8 and Figure 5.9. In Figures 5.8 and 5.9, the red node denotes the current location of the process, and the blue node denotes an unvisited neighbor. The yellow node means a visited node, and the black edge represents an edge of a spanning tree. The small number on each node denotes corresponding node index. The underlined number in the NL denotes the index of the currently selected node. In Figures 5.8 and 5.9, when the process selects a node from NL, we used mixed behaviors in which the process selects the most recently added node first and

a node at random next. Figures 5.8(g), 5.8(k), 5.8(n), 5.8(q), 5.8(t), 5.8(w) and 5.9(a) show cases in which the process selects the most recently added node, the right-most node index in NL, and Figures 5.8(d), 5.8(i), 5.8(m), 5.8(p), 5.8(s), 5.8(v), 5.8(y), and 5.9(c) shows the cases in which the process selects a node randomly in NL. Please note that, as shown in Figure 5.8(p), when the red node 5 does not have unvisited neighbors, the node 5 is removed from NL. An example of a spanning tree of the growing tree algorithm is represented in Figure 5.9(e).

5.1.8 Eller's algorithm

This algorithm processes one row at a time in a grid. In the algorithm, each node is assigned to its own set at the beginning, and, when nodes are connected by edges, the nodes are merged to be in the same set. When the algorithm processes a row, neighboring nodes that are in different sets are selected randomly and joined into the same set, and an edge between those nodes become an edge of a spanning tree. If nodes, which are already in the same set, are joined, it will form a loop in the resulted graph, so only nodes in different sets are joined. From each set of joined nodes, a downward adjacent vertical edge, e, is selected randomly and becomes an edge of the spanning tree. The bottom adjacent node of the edge e is joined into the same set of the top adjacent node of the edge e. Each set of nodes in the current row should have at least one downward vertical edge. For example, if the current row has three sets of nodes, at least one vertical edge should be selected from each set, so the number of selected vertical edges should be at least three. Otherwise, there is no connection to the set in the lower part of the graph, and the resulted graph will be disconnected.

The process starts from the top-most row where each node in the row is assigned to its own set initially. The process moves by a row. When the process arrives at the bottom-most

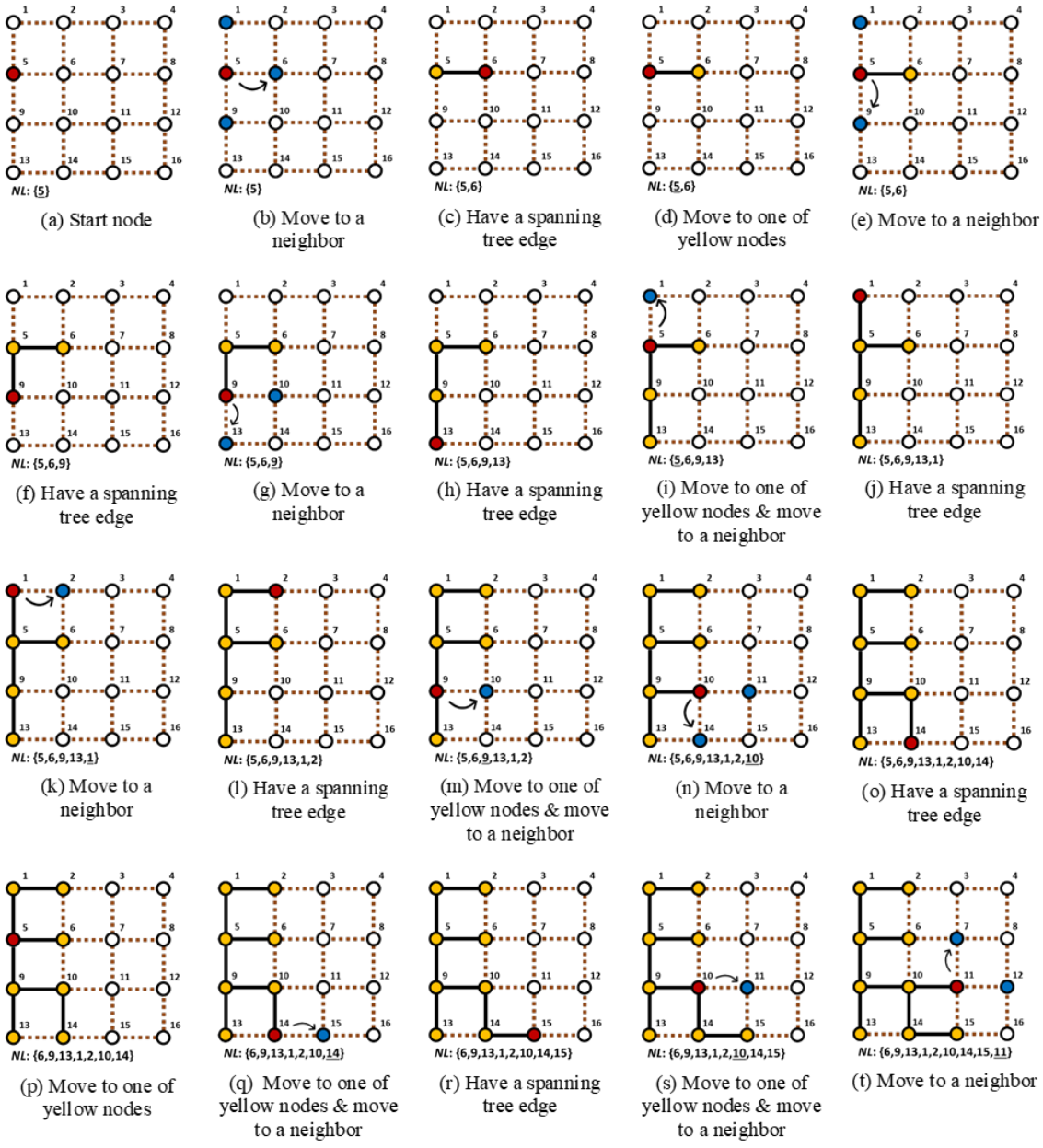


Figure 5.8: The progress of the growing tree algorithm.

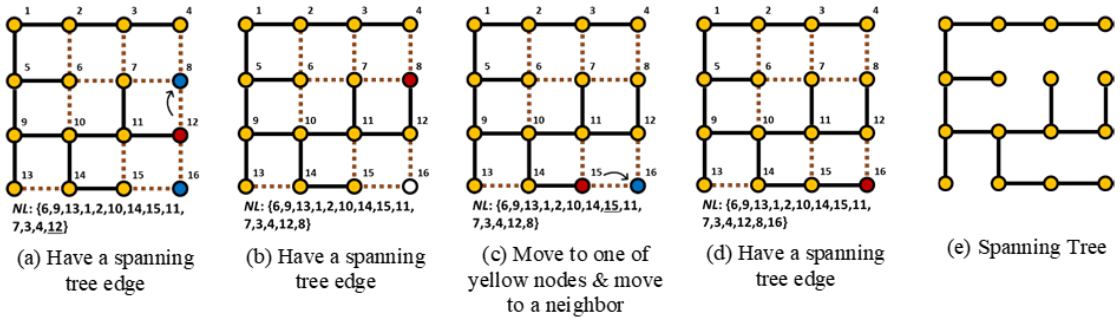


Figure 5.9: The progress of the growing tree algorithm (continued).

row in a grid, if nodes in the bottom-most row are in different sets, it means that they are not connected over a grid, and the resulted graph is not a spanning tree. Thus, all nodes in the bottom-most row need to be joined into a single set. In the bottom-most row, edges are placed between nodes of different sets so that no node is isolated in different set. Then, the process ends with a spanning tree.

The progress of Eller's algorithm is described in Figure 5.10. The color of each node denotes its corresponding set, so, if nodes have the same color, the nodes are in the same set. Please note that white node is not assigned to any set yet. The black edge denotes an edge of the spanning tree. As shown in Figure 5.10(a), nodes in the top row are initialized in such a way that each node has its own color. In an MxN grid, the top row has M colors initially. Then, neighboring nodes with different colors are selected randomly and connected with black edges as shown in Figure 5.10(b). Connected nodes now have the same color. As shown in Figure 5.10(c), from each color set, at least one of the downward adjacent vertical edges is randomly added to the spanning tree. Then, as shown in Figure

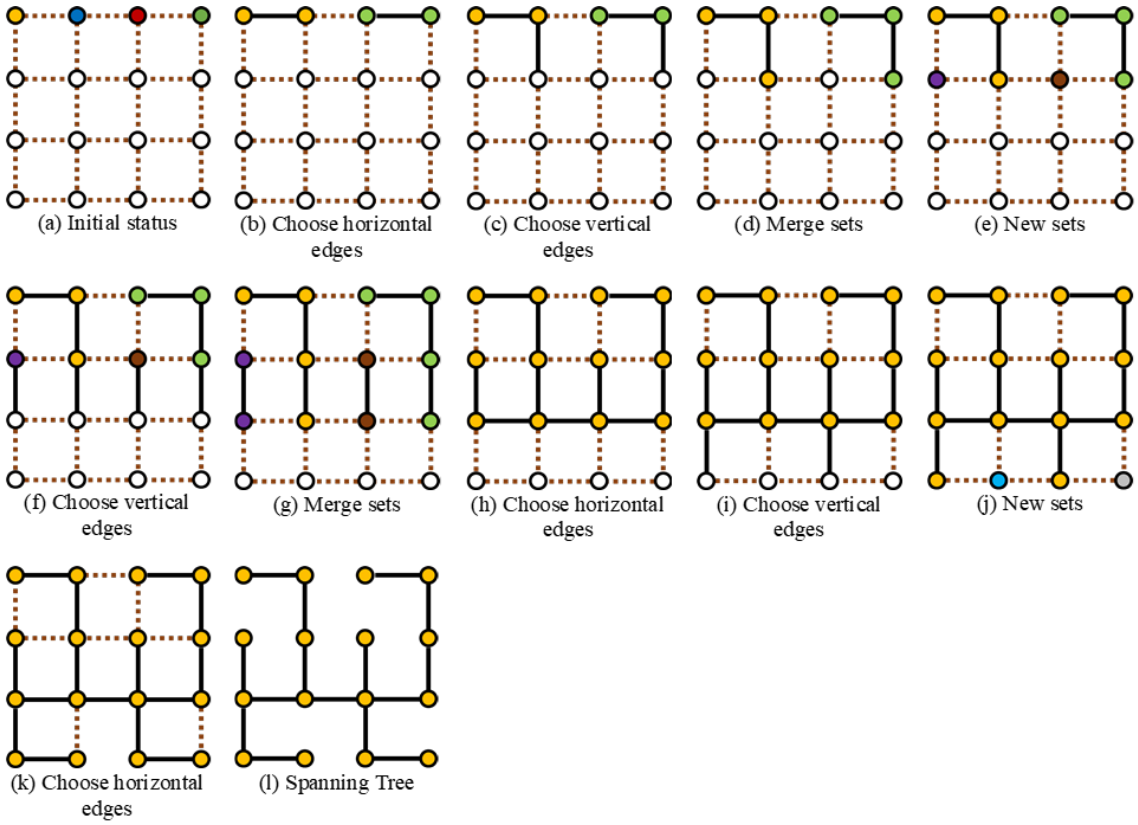


Figure 5.10: The progress of Eller's algorithm.

5.10(d), the bottom adjacent node of the black vertical edge becomes the same color of the connected upper node. As shown in Figure 5.10(e), nodes that are not connected to their upper neighboring nodes have their own color. In Figure 5.10(f), no nodes in the second row are connected horizontally. In this case, all vertical edges between the current row and the next lower row are selected as edges of a spanning tree as shown in Figure 5.10(g). As shown in Figures 5.10(j) and 5.10(k), at the last row, neighboring nodes which do not have the same color are connected so that all nodes in a grid are connected as a spanning tree. The algorithm ends with a spanning tree as represented in Figure 5.10(l).

5.1.9 Aldous-Broder algorithm

The process of Aldous-Broder algorithm [8] [13] starts at any node. Then, it visits one of its neighbors randomly, and the next action is determined based on the visited status of the neighbor. If the neighboring node has not been visited yet, that node and the prior node are connected by an edge of the spanning tree. If the neighboring node has been visited, the process simply moves to the next neighbor. Then, it moves to a random neighbor again. The process runs until all nodes are visited. Note that there is a case that this algorithm never ends because the last few nodes are not found randomly. When all nodes are visited, the process ends with a spanning tree.

The progress of Aldous-Broder algorithm is represented in Figure 5.11. The red node denotes the current location of the process, and the blue nodes denote candidates that the process is going to visit next. The yellow nodes are visited nodes, and the black edges are edges of a spanning tree. As shown in Figures 5.11(e), 5.11(f), and 5.11(g), even if a node is a yellow node, the node becomes blue node. If the red node moves to one of blue nodes which was not a yellow node before, the edge in which the red node passed becomes a black edge. As shown in Figures 5.11(g), 5.11(h), and 5.11(i), if the red node moves to one of blue nodes which was a yellow node before, the edge in which the red node passed does not become a black edge. The process results in a spanning tree as shown in Figure 5.11(w).

5.1.10 Wilson's algorithm

At the beginning of Wilson's algorithm [37], a node u of a grid graph is picked and marked as visited initially. Then, this algorithm picks a node v among any unvisited nodes and makes a path from the node v to the node u by randomly walking on the graph. While it is making a path, the nodes of the path, including the start point v, are not marked as

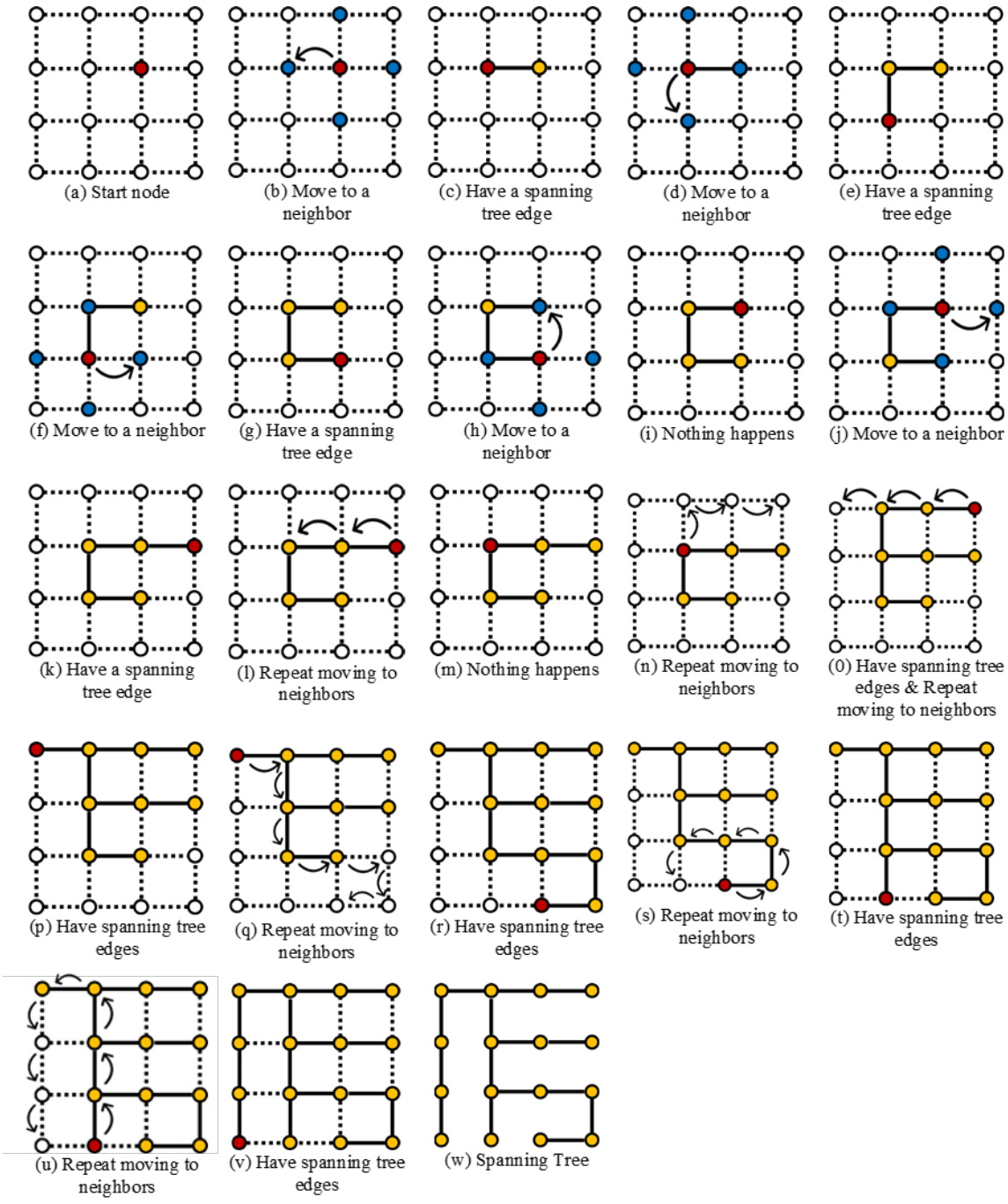


Figure 5.11: The progress of Aldous-Broder algorithm.

visited. The path making is stopped when it meets the node u. Then, all nodes of the path are marked as visited, and edges of the path become edges of the spanning tree.

During the path making, when the path forms a loop by encountering itself, the nodes of the loop, except the encountered node, are discarded from the path before going further. For example, if a path has a loop and consists of nodes 1 2 3 4 5 3, nodes 4 5 are removed from the path, and nodes 1 2 3 become a path. Then, the process continues to move randomly until it meets a node that is marked as visited. This process ensures the creation of a path without a loop.

After making a path that ends at the node u, and after all nodes of the path are marked as visited, the algorithm picks any node among the unvisited nodes and starts to make another path to one of the visited nodes. After all nodes have been marked as visited, a spanning tree is obtained as a result.

The progress of Wilson's algorithm is described in Figure 5.12. In Figure 5.12, the red, blue, and yellow nodes denote the current location of the process, candidates in which the process will visit next, and visited nodes, respectively. The process creates a path with purple edges, and, when the red node meets one of yellow nodes, purple edges become black edges which are edges of a spanning tree. Please note that when the path with purple edges creates a loop as shown in Figure 5.12(m), the loop with edges 1, 2, 3, and 4 are removed from the purple path as shown in Figure 5.12(n), and the process continues to make a path as represented in Figure 5.12(o). An example of a spanning tree generated by this algorithm is represented in Figure 5.12(z).

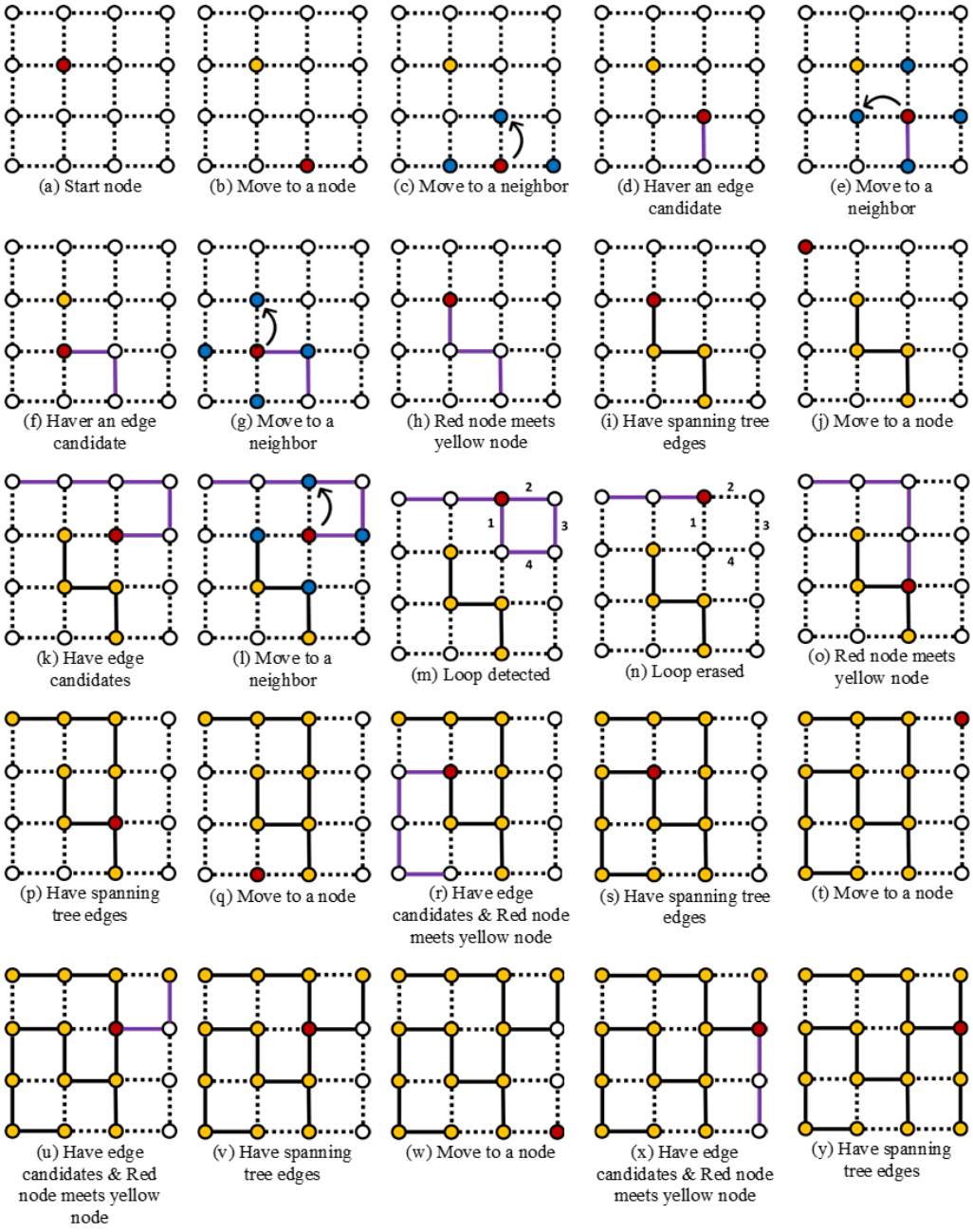


Figure 5.12: The progress of Wilson's algorithm.

5.2 Minimum Spanning Tree (MST) Algorithm

When there are weights on all edges of a grid graph, the MST algorithm can be used to generate a spanning tree. The MST algorithm finds a spanning tree in which edges of the spanning tree have total minimum weights.

Initially, we need an empty edge list EL. Then, we select edges of a grid graph based on their weights and store them in EL unless the edges in EL form a loop with the selected edges. The edge with the lowest weight is chosen first, and the edge with the highest weight is chosen last. The selection process is repeated until $M * N - 1$ edges are stored in EL from an MxN grid. $M * N - 1$ is the total number of edges of a spanning tree in an MxN grid structure, as explained in Chapter 3. The resulting graph in EL is the minimum spanning tree given the weights.

Figure 5.13 shows how MST algorithm works on a grid with the weights. In Figure 5.13, the small number on an edge denotes the weight on the edge. The purple edge is an edge selected by the process, and the black edge is an edge of a spanning tree. In Figure 5.13, we can see that the order of selecting edges as purple edges depends on their weights. When a purple edge, whose weight is 14, forms a loop with black edges as shown in Figure 5.13(v), the purple edge does not become a black edge as shown in Figure 5.13(w). Otherwise, a purple edge becomes a black edge. Figure 5.13(z) shows an example of a spanning tree generated by this algorithm.

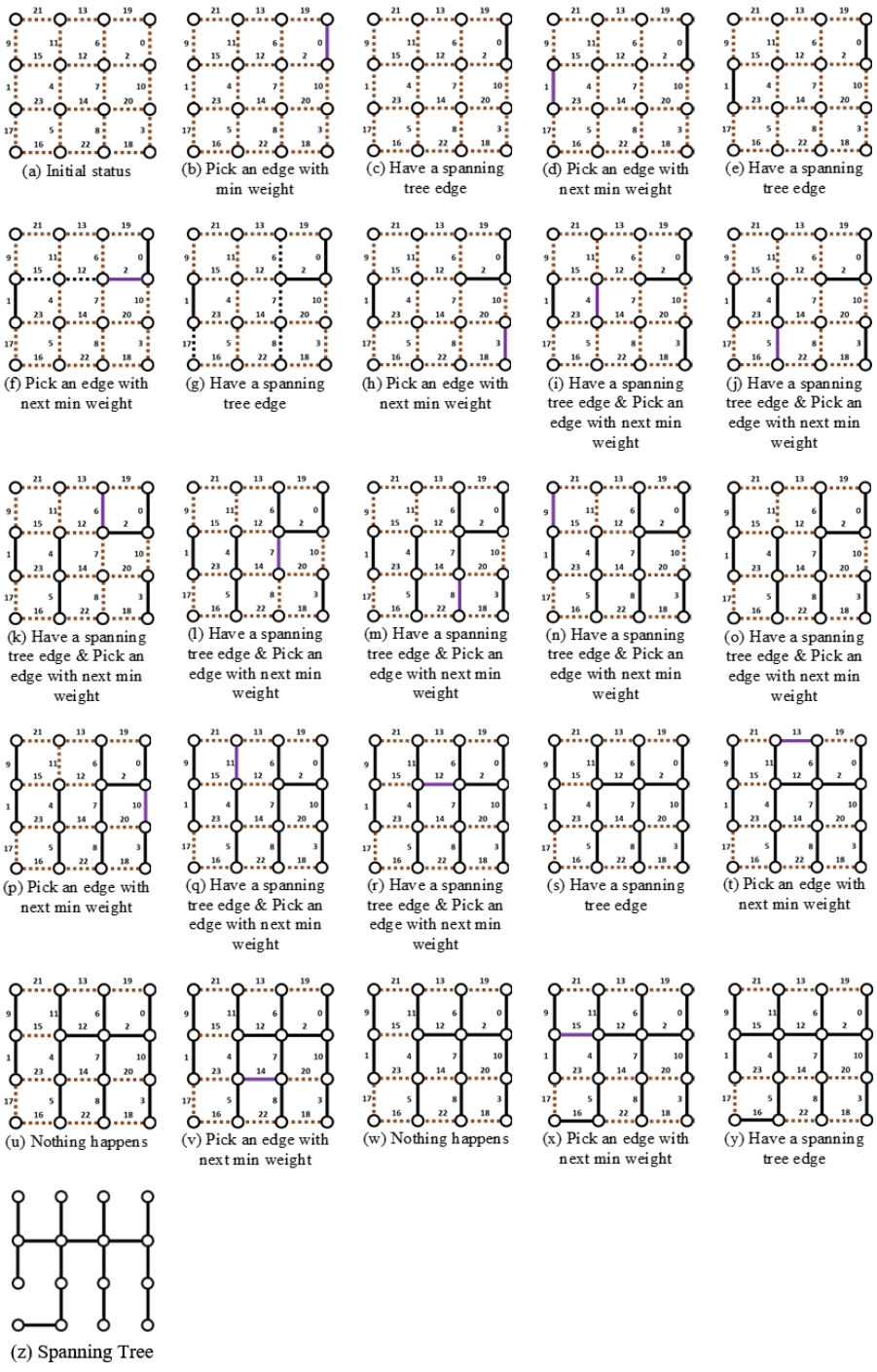


Figure 5.13: The progress of MST algorithm.

Chapter 6: Maze Representation

In our research, to represent a maze, we apply an idea of characteristic vector which shows whether or not specific items are included in a given structure as described in [33] and [29]. In this chapter, we explain our representation of a maze, which represents existence of edges in a grid.

6.1 Grid

Since a platform for our maze is a regular grid, to explain our maze representation, we provide a description of the grid first. An MxN grid has rectangular cells arranged in M columns and N rows. Each cell center is called a node, and each line segment between the nodes is called an edge. In an MxN grid, we have $M * N$ nodes and $M(N-1) + N(M-1)$ edges.

In Section 6.4, we will use concepts of a column and a row of a grid. Columns and rows denote sub-regions of a grid. Each column and each row have corresponding vertical edges and horizontal edges as shown in Figures 6.1 and 6.2. Note that all columns except the rightmost column have corresponding vertical and horizontal edges, but the rightmost column has vertical edges only. Likewise, all rows except the bottom-most row have corresponding vertical and horizontal edges, but the bottom-most row has horizontal edges only.

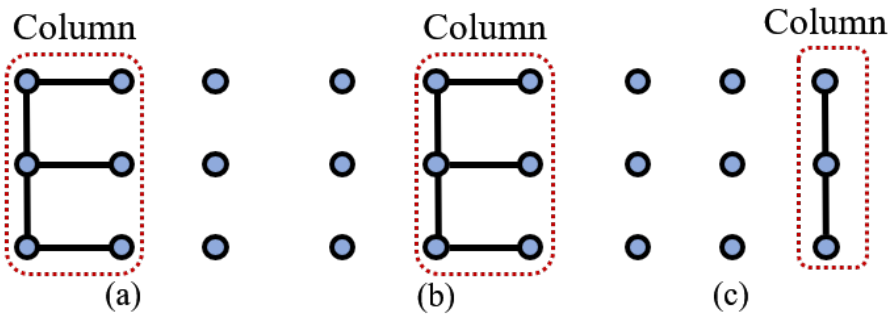


Figure 6.1: Figures show corresponding vertical and horizontal edges of each column on a 3x3 grid. (a) Edges of the leftmost column. (b) Edges of the second left column. (c) Edges of the rightmost column. Note that the rightmost column has only vertical edges.

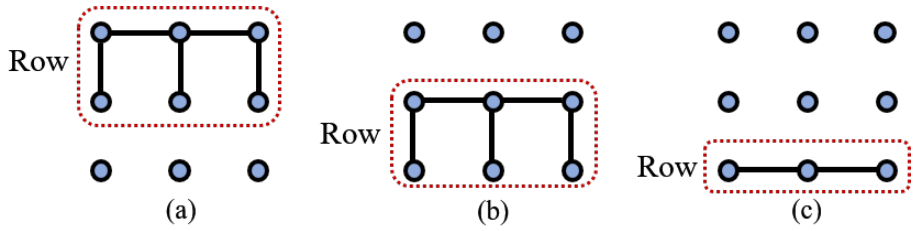


Figure 6.2: Figures show corresponding vertical and horizontal edges of each row on a 3x3 grid. (a) Edges of the topmost row. (b) Edges of the second top row. (c) Edges of the bottom-most row. Note that the bottom-most row has only horizontal edges.

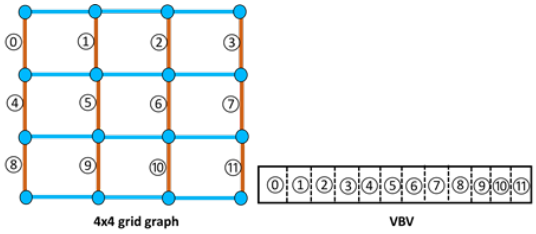
6.2 Edge Bit Vector on a Grid

In our research, a maze is a spanning tree on a grid, and our representation of a maze, called an edge bit vector (EBV), represents spanning tree edges on a grid. So, EBV shows which edges of a grid are selected as spanning tree edges. In the following sections, we explain this EBV with more details.

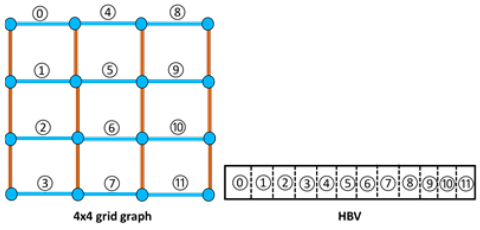
6.3 Edge Bit Vector (EBV)

In our research, as mentioned in Section 6.2, we represent a spanning tree in a computer using an edge-based representation, called an edge bit vector (EBV). Our EBV is a bit vector in which each bit tells if an edge of the grid exists in a spanning tree. A bit value of 1 denotes that the edge is in a spanning tree, and a 0 bit value denotes that the edge does not exist in a spanning tree. Our EBV has two components, a Vertical Bit Vector (VBV) and a Horizontal Bit Vector (HBV). Each bit in the VBV shows if a corresponding vertical edge exists in the spanning tree, and each bit in HBV shows if a corresponding horizontal edge exists in a spanning tree. In an MxN grid, VBV has M*(N-1) bits, and HBV has N*(M-1) bits.

As indicated in Figure 6.3, we can simply organize the edges of a grid into VBV and HBV separately. In our organization, going left to right in the VBV goes first left to right the topmost row and then to the left of the next row. Likewise, going left to right in the HBV goes first down the leftmost column and then similar for each column. For reference, in Figure 6.4, we illustrate how a spanning tree is represented by VBV and HBV using this organization. This column and row layout of bits will prove advantageous in our new spanning tree enumeration algorithm in Chapter 8.



(a) Relationship between vertical edges of 4x4 grid graph and VBV.



(b) Relationship between horizontal edges of 4x4 grid graph and HBV.

Figure 6.3: Vertical Bit Vector (VBV) and Horizontal Bit Vector (HBV) and their corresponding edges in the graph. In a grid, vertical edges are denoted by brown edges, and horizontal edges are denoted by blue edges.

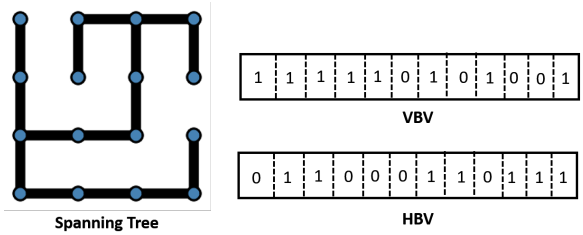


Figure 6.4: Example showing the spanning tree and its corresponding VBV and HBV.

6.4 Constraints on EBV

When we create a bit vector in EBV to represent a spanning tree, not all bit combinations of the EBV yield a spanning tree. For example, when all bits of an EBV are 1 bit, the EBV represents the complete grid or graph. There are several constraints which need to hold on the EBV to produce a spanning tree. In this section, we show which constraints the EBV must have.

To represent the constraints easier, we define several sets below first.

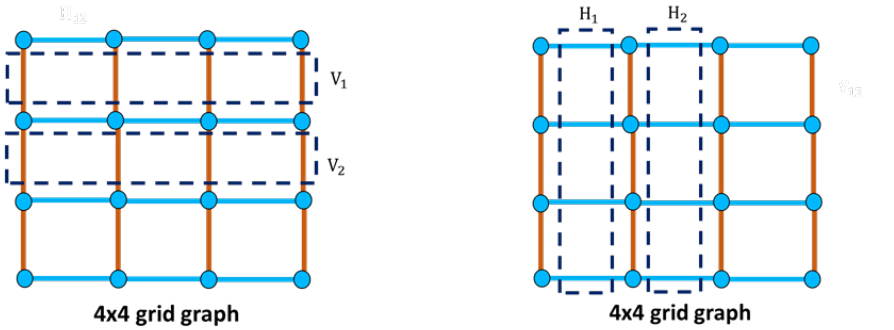
V: Set of 1 bits (or edges) of the VBV. Number of elements of this set equals to the number of 1 bits in the VBV.

H: Set of 1 bits (or edges) of the HBV. Number of elements of this set equals to the number of 1 bits in the HBV.

V_i: Set of 1 bits of the VBV between i$^{\text{th}}$ row and the next row, (i+1)$^{\text{th}}$ row, in an MxN grid, where $1 \leq i \leq N-1$. Number of elements of this set equals to the number of 1 bits in the VBV between i$^{\text{th}}$ row and (i+1)$^{\text{th}}$ row. For example, as shown in Figure 6.5(a), V_1 is corresponding to 1 bits between the first row and the second row in a grid, and V_2 is corresponding to 1 bits between the second row and the third row in a grid.

H_j: Set of 1 bits of the HBV between j$^{\text{th}}$ column and the next column, (j+1)$^{\text{th}}$ column, in an MxN grid, where $1 \leq j \leq M-1$. Number of elements of this set equals to the number of 1 bits in the HBV between j$^{\text{th}}$ column and (j+1)$^{\text{th}}$ column. For example, as shown in Figure 6.5(b), H_1 is corresponding to 1 bits between the first column and the second column in a grid, and H_2 is corresponding to 1 bits between the second column and the third column in a grid.

Now, we show the constraints on the EBV in an MxN grid.



(a) Bits of vertical edges corresponding to V_i. (b) Bits of horizontal edges corresponding to H_j.

Figure 6.5: Representation showing bits of edges in a grid corresponding to V_i and H_j. V_i and H_j contains 1 bits of VBV between corresponding rows and 1 bits of HBV between corresponding columns, respectively.

1. $|V| + |H| = M \times N - 1$

   Since a grid has $M * N$ nodes, this constraint is derived from a property of a spanning tree in which a spanning tree has $M \times N - 1$ edges. If there are not $M \times N - 1$ in a graph, the graph is not a spanning tree. Corollaries also exist, such as $|H| = M * N - 1 - |V|$ and etc.

2. For all i, $|V_i| \geq 1$

   A spanning tree is a connected graph in which all nodes of a grid graph are connected via its edges. Thus, a spanning tree has at least one vertical edge that connects nodes in one row to nodes in another row. For example, as shown in Figure 6.6(a), if there is no vertical edge between rows, even if all nodes are connected by horizontal edges, it cannot be a connected graph. In terms of VBV, there needs to be at least one 1 bit in the VBV between rows in a grid. Since $|V_i|$ denotes number of 1 bits in VBV between rows in a grid, $|V_i|$ should be greater than 0.

3. For all j, $|H_j| \geq 1$

   This is similar with the second constraint. A spanning tree has at least one horizontal edge that connects nodes in one column to nodes in another column. An example of a disconnected graph is shown in Figure 6.6(b). In terms of HBV, there should be at least one 1 bit in the HBV between columns in a grid. Since $|H_j|$ denotes number of 1 bits in HBV between columns in a grid, $|H_j|$ should be greater than 0.

4. $|V| \geq N - 1$

   According to the second constraint, VBV has at least one 1 bit between i^{th} row and $(\mathrm{i}+1)^{\mathrm{th}}$ row in a grid, and, since $1 \leq i \leq N - 1$, VBV has at least $N - 1$ 1 bits.

5. $|H| \geq M - 1$

   This constraint is similar to the fourth constraint but it is related to 1 bits in HBV. From the third constraint, HBV has at least one 1 bit j^{th} column and $(\mathrm{j}+1)^{\mathrm{th}}$ column in a grid, and, since $1 \leq j \leq M - 1$, HBV has at least $M - 1$ 1 bits.

The above constraints are necessary conditions of EBV to create a spanning tree. However, an EBV that satisfies the above constraints do not guarantee to provide a spanning tree.

6.5 Metric Computation from EBV

When we analyze the spanning tree space later in Chapter 10, we evaluate maze metrics on a spanning tree. Our EBV representation does not bring an advantage on a performance of the evaluation. But, In our research, since a spanning tree is represented by an EBV, we show how to compute the basic maze metrics (#Turns, #Straights, #T-Junctions, #Cross-Junctions, #Terminals) from VBV and HBV.

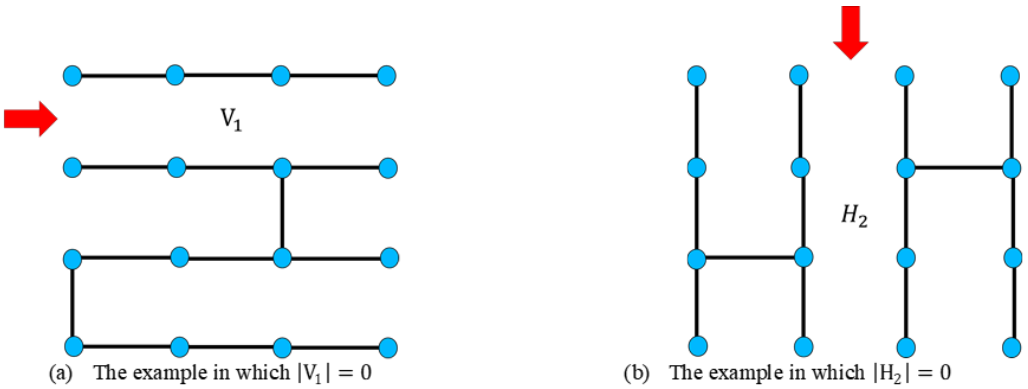


Figure 6.6: Examples of a disconnected graph.

As explained in Chapter 4, since the basic maze metrics are measured by counting cells of corresponding type on the spanning tree, we need to know what kind of cell type each node has on a spanning tree. Before we explain metric computation from VBV and HBV, we show how a cell type of a node is determined from the VBV and HBV. First, four bits corresponding to four adjacent edges of a node are extracted from the VBV and HBV as shown in Figure 6.7. Two bits in VBV correspond to top edge and bottom edge of the node, and two bits in HBV correspond to left edge and rights edge of the node. The four bits are extracted and gathered into a four-bit bit vector in consecutive order. Then, using the dictionary provided in Table 6.1, a bit vector of these four bits are converted to the corresponding cell type. Note, the entry $0, 0, 0, 0$ is missing from the table, as this would lead to an orphaned node not in the spanning tree.

Next, we show how to compute metrics from a VBV and HBV. Initially, we set all metric values as 0. Then, when we compute metric values of a spanning tree, we iterate over nodes on the spanning tree. And, at each node, when a cell type of the node is determined from

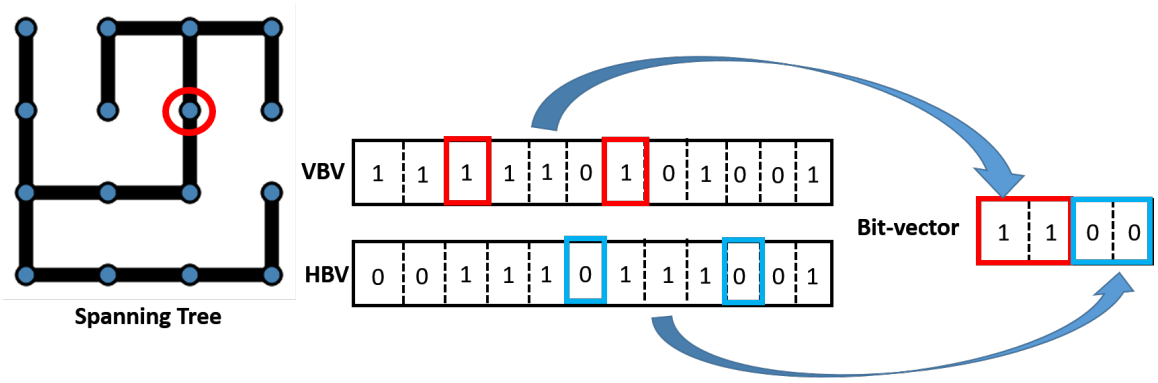


Figure 6.7: Example of making a bit vector that representing adjacent edges of a node in a spanning tree. In this example, we look at the node denoted by the red circle in the spanning tree. Two bits in VBV, denoted by red rectangles, and two bits in HBV, denoted by blue rectangles, create a bit vector of the node's neighbors.

Bit vector	**Cell type**
{1,0,1,0}	Turn
{1,0,0,1}	Turn
{0,1,1,0}	Turn
{0,1,0,1}	Turn
{0,0,1,1}	Straight
{1,1,0,0}	Straight
{1,0,1,1}	T-Junction
{1,1,0,1}	T-Junction
{1,1,1,0}	T-Junction
{0,1,1,1}	T-Junction
{1,1,1,1}	Cross-Junction
{0,0,1,0}	Terminal
{1,0,0,0}	Terminal
{0,1,0,0}	Terminal
{0,0,0,1}	Terminal

Table 6.1: Dictionary to convert bit vector of four bits to cell type.

VBV and HBV using Table 6.1, we raise the corresponding metric value by 1. For example, if the cell type of the current node is Turn, the value of the metric #Turns is raised as #Turns + 1.

In Algorithm 1, we provide pseudocode to illustrate how to measure metrics from EBV. In Algorithm 1, BitVecFromNode is a function in which a node index of a spanning tree, VBV, and HBV are inputs, and a four-bits bit-vector representing adjacent edges of the input node is returned as depicted in Figure 6.7. CellTypeDict is the dictionary represented in Table 6.1. When a bit vector of four-bits inputs to CellTypeDict, cell type corresponding to the input is returned using Table 6.1. MetricValDict is a dictionary where a key is a cell type and a value is a value of corresponding metric. If Turn is input as a key to MetricValDict, the value of the metric #Turns is returned.

Algorithm 1 iterates over all nodes of a spanning tree with given VBV and HBV. For each node, the process obtains corresponding four-bits bit vector from VBV and HBV using BitVecFromNode. The obtained bit-vector inputs to CellTypeDict, and corresponding cell type is returned. The returned cell type is input to MetricValDict, and the metric value corresponding to input cell type is raised. After the iteration over all nodes is finished, the algorithm stops and MetricValDict is returned. The returned MetricValDict includes values of all metrics measured from given VBV and HBV.

Algorithm 1 Measure metrics from EBV in a MxN grid graph

1: **function** MetricValuesOnSpanningTree(VBV, HBV)
2: *Initialize* $BitVec$, $CellType$, $MetricValDict$
3: **for** $Node \leftarrow 1$ to $M * N$ **do** ▷ $M * N$: Number of nodes in a spanning tree
4: $BitVec \leftarrow BitVecFromNode(Node, VBV, HBV)$
5: $CellType \leftarrow CellTypeDict[BitVec]$
6: $MetricValDict[CellType] \leftarrow MetricValDict[CellType] + 1$
7: **end for**
8: **return** $MetricValDict$
9: **end function**

Chapter 7: Search-based procedural content generation

In Chapter 5, we presented current existing spanning tree generation algorithms. All of these algorithms use a pseudo random number as an input parameter. This input parameter gives us a lack of control over the generated spanning tree. Since we cannot effectively control the spanning tree generation in the general case, we are forced to resort to iterative random generation to find a spanning tree with desired properties. In this chapter, we introduce the iterative random generation approach to generate a spanning tree based on desired properties. The approach is called search-based procedural content generation.

7.1 Procedural Content Generation for Games

Before explaining search-based procedural content generation, we explain procedural content generation for games (PCG) first. As described in [34], the procedural content generation is related to creation of game content using algorithms with or without user input. The algorithms used in PCG generates various shapes and/or functions of content based on a few input parameters. A domain of game content can be weapons, dungeons, game rules, terrains, or items. Several commercial games such as Diablo, Spore, Minecraft, and Elder Scroll have utilized this PCG technique.

However, often PCG is not easily controlled. Sometimes it is hard to know how to manipulate input parameters to obtain desired outputs on the fly. Thus, we apply SBPCG

approach which repeatedly generate-and-test contents until the process satisfies a given criteria.

7.2 Search-based Procedural Content Generation

Search-based procedural content generation (SBPCG) is one of the approach to PCG. The process of SBPCG approach is illustrated in Figure 7.1. As shown in Figure 7.1, when desired properties are given, SBPCG searches contents by generating ones over the search space. The search space is the space of content. During searching on the search space, evaluation function, called fitness function, evaluates a found content to check whether the content satisfies given inputs. The content which has properties close to desired properties is given higher score by the fitness function. The searching process ends when it meets given criteria. Termination criteria can vary and can be defined by the user. For example, the searching process can end when it meets the certain number of searches or when the certain number of desired contents found. After the searching process stops, contents with the highest score are returned as the best desired contents.

7.3 SBPCG in Finding Desired Maze

In this section, we show a framework of SBPCG applied in generation of a maze, which is our content domain. Figure 7.2 depicts how we find mazes based on desired properties using SBPCG approach. Desired properties are represented by maze metrics. As shown in Figure 7.2, in searching process of desired mazes, we search/generate spanning trees over the search space. The search space is ideally the whole space of spanning trees. In a brute force approach to explore the search space, we randomly generate spanning trees. In latter chapters, we will also introduce intelligent searching techniques to reduce search time. When a spanning tree is generated, metric values are measured on the spanning tree, and

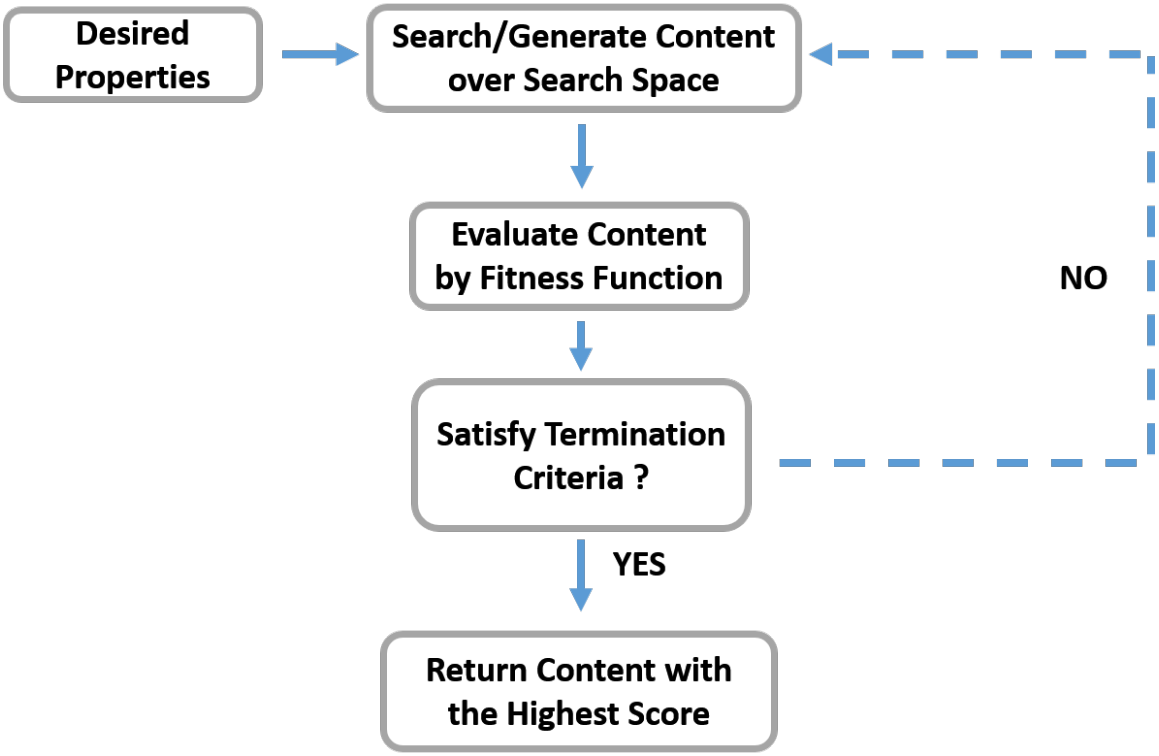


Figure 7.1: Flow chart of SBPCG approach.

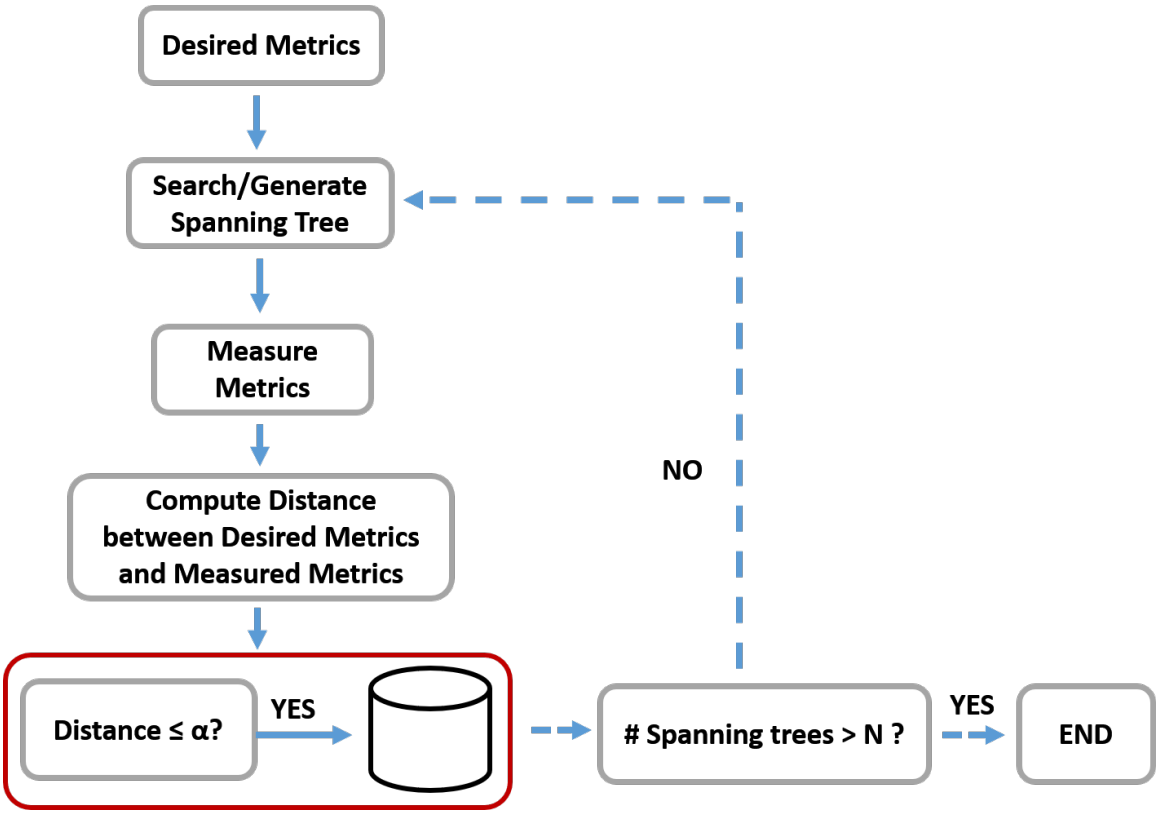


Figure 7.2: Flow chart of desired maze generation using SBPCG concept.

the process computes a distance between a vector of measured metrics (#Turns, #Straights, #T-Juncs, #Cross-Juncs, #Terminals) and a vector of given desired metrics (#Desired-Turns, #DesiredStraights, #DesiredT-Juncs, #DesiredCross-Juncs, #DesiredTerminals). If the distance is less than given threshold α, the spanning tree is considered as desired spanning tree and stored. This searching process repeats until the process stores N spanning trees. When the process stops, the stored spanning trees are returned as possible candidates.

7.4 Metrics to evaluate performance of SBPCG-based method

In this section, we introduce metrics to evaluate a performance of SBPCG-based method, which uses SBPCG approach to find desired mazes based on given inputs. The metrics show whether the corresponding method is capable to find desired mazes satisfactorily.

Time to Search:This metric measures time to find N spanning trees satisfying given input parameters.

Capability of search: This metric investigates if the method searches spanning trees on the right spot of a search space. For example, if it searches spanning trees with #Terminals=2 when it is in the area of spanning trees #Terminals=20, we can say that the method has no capability of search. The metric counts how many desired spanning trees are found while M spanning trees are sampled. If few desired spanning trees are found within M sampled spanning trees, then the corresponding method has a low capability of search. If many desired spanning trees are found within M sampled spanning trees, then the corresponding method has a high capability of search.

Chapter 8: Spanning Tree Enumeration with Edge Bit Vectors

In Chapter 5, we introduced existing spanning tree algorithms. Some of the current algorithms do not generate spanning trees equally and have different characteristics over the sampled spanning trees. For example, as described in [32] and [15], the recursive backtracker is known to have a high probability to generate spanning trees with long passages and small branches as shown in Figure 8.1(a), and Prims algorithm is known to have a high probability to generate a spanning tree with numerous branches as shown in Figure 8.1(b).

These algorithms can be used in SBPCG approach to find spanning trees with desired properties. However, if we do not know their different characteristics over the resulting spanning trees in detail, it is hard to choose the appropriate algorithm to provide spanning trees that we want. To figure out these characteristics of the algorithms in detail, we compare the sampling spaces of the algorithms to the enumerated space of all spanning trees. There are several works that analyze some maze generation algorithms ([24], [20]), but they do not compare the sampling spaces to the enumerated space. Our comparison allows us to know what kind of spanning trees in the enumerated space are generated with difficulty or easily by each algorithm. For this comparison, we want a gold standard to which all spanning tree generation algorithms can be compared, and in order to obtain a gold standard, we enumerate all spanning trees.

Figure 8.1: (a) A spanning tree sampled easily using the recursive backtracker algorithm. (b) A spanning tree sampled with difficulty using the recursive backtracker algorithm.

However, enumeration is an NP problem. Solving the problem is not a trivial task and requires exhaustive searches. Suppose that we enumerate spanning trees using the minimum spanning tree (MST) algorithm [18]. In this enumeration process, first, we generate all possible combinations of weight values for edges of a grid. Weight values are not duplicated, so a unique MST is generated for each combination of weight values. For each combination of edge weight values, we use the MST algorithm to generate a spanning tree. After generating spanning trees for all possible combinations of edge weight values, all possible spanning trees can be obtained. If we use Kruskal's algorithm to find an MST, the running time for generating an MST for each combination of edge weights is $O(M * N * \log(M * N))$ in an MxN grid. Since there are $2 * M * N - M - N$ edges in an MxN grid, there are $(2 * M * N - M - N)!$ possible combinations of edge weights, and the total running time for this brute-force method is $O((M * N)! * M * N * \log(M * N))$

With this brute-force method, tremendous work is required to generate all possible combinations of edge weight values. For example, in a 6x6 grid, there are $60! \approx 8.32 * 10^{81}$ combinations. This brute-force method gives us duplicate spanning trees. Even if orders

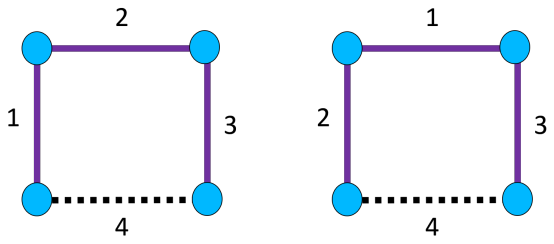


Figure 8.2: Examples of duplicated MSTs on different combinations of edge weight values in a 2x2 grid. Each number denotes weight value of a corresponding edge, and the purple line denotes the edge of the spanning tree.

of the edge weight values are different on a grid, we can have duplicate MSTs, as shown in Figure 8.2. In Figure 8.2, two 2x2 grids have a different order of weight values, but they have the same MST. Thus, to remove duplicates, we would need to compare all $8.32 * 10^{81}$ spanning trees. Even in a small size of a 6x6 grid, and it is not feasible to do so. One way to optimize the enumeration process is by adding constraints to control the process so that the number of operations can be reduced. However, because there is no freedom to adding constraints with this method, it is also hard to optimize. As we can see with this method, the enumeration process would require a huge amount of work and, in reality, it is not possible.

In this chapter, we introduce a new enumeration method, which makes the enumeration process more feasible by using Edge Bit Vectors (EBV).

8.1 Enumeration with Vertical Bit Vector and Horizontal Bit Vector

As explained in Chapter 6, EBV is a bit vector, with two components: a Vertical Bit Vector (VBV) and a Horizontal Bit Vector (HBV). VBV and HBV respectively show the existence of vertical edges and horizontal edges on a spanning tree using bit vectors.

In a 6x6 grid, given that there are 60 edges, the number of all possible bit patterns for VBV and HBV is $2^{60} \approx 1.15 * 10^{18}$, which is much less than the number of all possible combinations of edge weight values, $8.32 * 10^{81}$. Furthermore, it is easy to add constraints to bits to control the bit pattern generation, such that the amount of exhaustive work can be reduced. Also, we can more easily avoid duplicates, as each EBV is a unique graph In the next sections, we show how we can enumerate spanning trees using VBV and HBV.

Before going further, we define some notations clearly so that the explanation about our enumeration process is more understandable.

Column i: In Chapter 6, column i of a grid contained only vertical edges, but we extend it to also contain horizontal edges in this chapter. As shown in Figure 8.3(a), in a grid, each column has corresponding vertical edges and horizontal edges. As shown in Figure 8.3(a), in an MxN grid, column 0 starts at the left-most column and column M-1 at the right-most column. Please note that, in the last column M-1, there are only vertical edges not horizontal edges.

VE_i: Vertical edges in column i. It is sub-bit-vector of VBV corresponding to vertical edges in column i as shown in Figure 8.3(b). In an MxN grid, there are M columns.

HE_i: Horizontal edges in column i. It is sub-bit-vector of HBV corresponding to horizontal edges in column i as shown in Figure 8.3(c). In an MxN grid, there are total M-1 HE_is. It is same with the notation H_i defined in Chapter 6, but, to maintain a unity with the notation VE_i, we write H_i as HE_i in this chapter.

8.2 Brute-force enumeration method using VBV and HBV

First, we demonstrate a simple way to enumerate spanning trees using VBV and HBV. In this enumeration process, we enumerate bit patterns on VBV, and, for each VBV, we

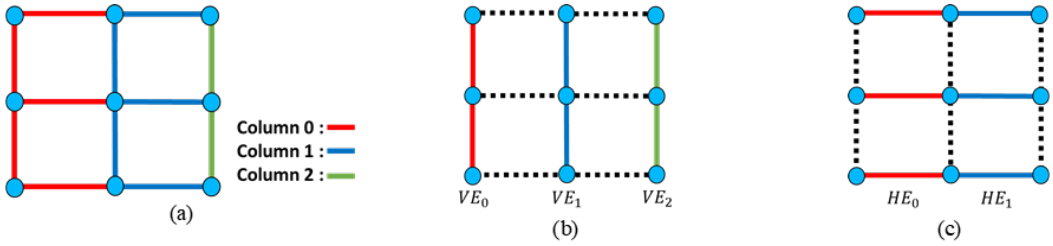


Figure 8.3: . (a) Figure of 3x3 grid representing corresponding vertical edges and horizontal edges of each column i. Edges of different columns are distinguished by different colors. (b) Figure of 3x3 grid representing VE_i. (c) Figure of 3x3 grid representing HE_i.

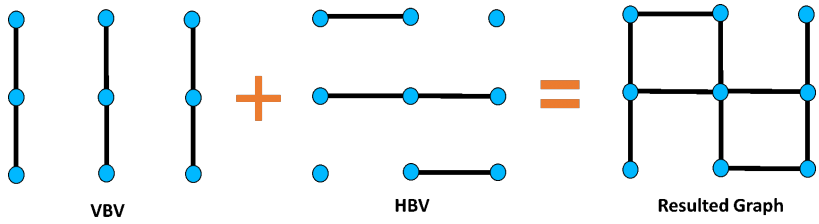


Figure 8.4: (a) Figure of 3x3 grid representing corresponding vertical edges and horizontal edges of each column i. Edges of different columns are distinguished by different colors. (b) Figure of 3x3 grid representing VE_i. (c) Figure of 3x3 grid representing HE_i.

enumerate bit patterns on HBV. After we have a pair of VBV and HBV, since not all pairs of VBV and HBV give a spanning tree, as shown in Figure 8.4, we need to investigate whether the pair yields a spanning tree by checking loops or disconnected components over a grid.

The pseudocode for this brute-force method is provided in Algorithm 2. In the pseudocode, VBVSet and HBVSet contain all possible bit patterns for VBV and HBV, respectively. For each VBV from VBVSet, we retrieve all HBVs from HBVSet. If a pair of VBV and HBV generates a spanning tree, they are returned as a spanning tree.

Algorithm 2 Iterative Policy Evaluation Algorithm

```
1: function BruteForceWithVBVandHBV
2:     for each VBV ∈ VBVSet do
3:         for each HBV ∈ HBVSet do
4:             if (VBV, HBV) produces a spanning tree then
5:                 Yield (VBV, HBV)
6:             end if
7:         end for
8:     end for
9: end function
```

Now, we analyze a running time for Algorithm 2. In Algorithm 2, the number of all possible bit patterns for VBV and HBV is $O(2^{M*(N-1)})$ and $O(2^{N*(M-1)})$, respectively. Thus, the running times for line number 2 and line number 3 are $O(2^{M*(N-1)})$ and $O(2^{N*(M-1)})$, respectively. In the checking process at the line number 4, the process investigates whether there are loops or disconnected components on VBV and HBV over a grid. Loops and disconnected components indicate that a pair of VBV and HBV does not produce a spanning tree. To check loops or disconnected components over a grid, we can use a depth-first search (DFS) approach [18]. During the DFS checking process, if a previously visited node, it denotes that a loop is found. If there are nodes on a grid where the DFS checking process did reach, then a disconnected component is found. Thus, the running time of the checking process with DFS can be $O(V + E)$, where V is the number of nodes, and E is the number of edges. Since $V = M * N$ and $E = 2 * M * N - M - N$ in an MxN grid, $O(V + E)$ is

changed to $O(M * N + 2 * M * N - M - N)$. Therefore, total running time of Algorithm 2 is $O(2^{M*(N-1)} * 2^{N*(M-1)}) * (M * N + 2 * M * N - M - N))$, which can be reduced to $O(4^{M*N} * M * N)$. By using VBV and HBV, the enumeration process becomes faster than the method with MST algorithm $O(4^{M*N} * M * N) < O((M * N)! * M * N * \log(M * N))$. Also, unlike the MST approach, it does not allow for duplicated spanning trees.

However, in this method there are always pairs of VBV and HBV which do not generate spanning trees, as shown in Figure 8.4. Hence, many invalid VBVs and HBVs, which do not produce a spanning tree, are generated unnecessarily and need to be filtered. This can be time-consuming. Thus, we introduce a new column-sweeping method, in which VBV and HBV are generated sequentially, and any invalid (VBV, HBV) pairs can be prevented before they are constructed.

8.3 Column-Sweeping Method with VBV and HBV

Using the previous brute-force method with VBV and HBV, it is hard to avoid invalid pairs of VBV and HBV before they are generated. Therefore, in this section we provide a new column-sweeping method, in which the process checks loops or isolated components locally rather than globally so that invalid VBV and HBV pairs are prevented in advance.

In this method, VE_i and HE_i are enumerated sequentially from the left column i to the right column i+1 of a grid. For example, after VE_i and HE_i are enumerated, VE_{i+1} and HE_{i+1} are enumerated based on each VE_i and HE_i. In this method, there are two cases, in which we consider VE_i and HE_i as invalid. As shown in Figure 8.5, one case involves having loops, and another case involves having isolated components over a grid. While we enumerates bit patterns on column i, we can check whether there are loops or isolated components between the first column and the current column. If loops or isolated

Figure 8.5: Examples showing invalid cases that we can have during the column-sweeping process are (a) an invalid case of having a loop and (b) an invalid case of having an isolated component. An isolated component is marked by a red-dashed box.

components are found, it enumerates a new VE_i and HE_i until it finds a valid one not producing invalid cases. If there is no valid one, then it backtracks to the previous column i-1 and reapplies other bit patterns of VE_{i-1} and HE_{i-1}. This backtracking process continues until loops or isolated components disappear on a grid. Like the previous brute force method with VBV and HBV, it can apply the DFS approach on columns between the first column and the current column in the checking process.

The pseudocode for this method is given in Algorithm 3. The program starts with colSweep(0, $\emptyset$, $\emptyset$). VBV and HBV are empty sets initially and combined with valid VE_i and HE_i during the process. In this pseudo code, VEiSet contains all possible bit patterns for VE_i, and HEiSet contains all possible bit patterns for HE_i. However, bit patterns in HEiSet also satisfies the constraint $|\text{HE}_i| \geq 1$, as explained in Chapter 6. When HE_i contains only 0 bits, it denotes that the graph is not connected on column i by HE_i, as described in Chapter 6. In the code, for each VE_i from VEiSet, we retrieve all HE_is from HEiSet and check which pair of VE_i and HE_i does not give loops or isolated components between the first column and the current column. If the pair does not give an invalid case, the process goes to the next column, in a recursive manner. When it comes to the

last column M-1, because there is only VE_i in the last column, it only checks if VE_{M-1} produces a spanning tree with the given VBV and HBV. If they produce a spanning tree, a pair of $VBV \cup VE_{M-1}$, along with HBV, is returned as a spanning tree of an enumerated set.

Algorithm 3 Column-Sweeping Enumeration Method using VBV and HBV

```
1: function colSweep(i, VBV, HBV)
2:     if i = M − 1 then
3:         for each VE_i ∈ VEiSet do
4:             if VE_i ∪ VBV and HBV are valid then
5:                 Return (VBV ∪ VE_i, HBV)
6:             end if
7:         end for
8:     else
9:         for each VE_i ∈ VEiSet do
10:            for each HE_i ∈ HEiSet do
11:                if VE_i and HE_i do not produce invalid cases with VBV and HBV then
12:                    colSweep(i + 1, VBV ∪ VE_i, HBV ∪ HE_i)
13:                end if
14:            end for
15:        end for
16:    end if
17: end function
```

Now, we analyze a running time for Algorithm 3. In the above pseudocode, the number of all bit patterns for VE_i is 2^{N-1}, and the number of all bit patterns for HE_i is $2^N - 1$ because the bit pattern with 0 bits is excluded for HE_i. Since the checking process applies DFS between the first column to column i, the running time of the checking process is $O(j * N + 2 * j * N - j - N)$, where $j = i + 1$. The running time can be reduced to $O(j * N)$. Thus, for each level of recursion, running time is $O(2^{N-1} * (2^N - 1) * j * N)$. When it comes to the last level of recursion ($i = M - 1$), since it checks only VE_i, the running time of the last level is $O(2^{N-1} * M * N)$. As there are $O(M)$ levels of recursion, including the last

level, the total running time is O($(2^{N-1} * (2^N - 1) * j * N)^{M-1} * 2^{N-1} * M * N$), which can be reduced to O($4^{N*M} * N^M * M!$).

Invalid VE_i and HE_i that produce loops or isolated components are disallowed locally, so there is no case in which we have invalid VBV and HBV in this method. However, the process still needs to check over a grid from the first column to the current column whenever VE_i and HE_i are generated, and that checking process results in a worse running time than the brute-force method with VBV and HBV. Although the running time is the upper bound of actual running time, and the actual running time can be lower, this time-consuming checking process is still not desirable. In the next section, we introduce an intelligent column-sweeping method, which enumerates spanning trees with VBV and HBV efficiently.

8.4 An Intelligent Column-Sweeping Method with VBV and HBV

Like the previous column-sweeping method, this method generates VE_i and HE_i based on information from the previous column i-1. However, instead of scanning a grid from the first column to the current column i to determine whether the generated VE_i and HE_i produce invalid cases, it goes through columns without any checking and only checks the last column to determine whether there are loops or isolated components between the first column and the last column. In the next section, we introduce our connected component concept, which makes this optimization possible.

8.4.1 Connected Components

As explained in [18], a connected component is a subgraph, in which nodes are connected to each other but not to any node in a supergraph, which is outside of the subgraph. For example, Figure 8.6 shows a graph with three connected components. Each connected

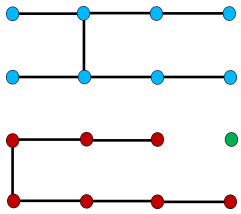


Figure 8.6: Graph with three connected components in a grid. The nodes of each connected component have their own color.

component is distinguished by a different color. In Figure 8.6, we can see that the green connected component contains a single node. When a node is not connected to any other node on a grid, the node is considered a connected component itself.

When we have different connected components on a grid, we distinguish them with a label by assigning each node a number called the component number (CN). Nodes with the same CN are in the same connected component, and nodes with different CNs are in different connected components.

In this section, we establish a new notation CN_i, for clarification.

CN_i: The vector of the CNs of the nodes in column i. For example, as shown in Figure 8.7, when the nodes in column i have CNs 0, 0, 0 and 1, CN_i is (0,0,0,1). Likewise, when the nodes in column i+1 have CNs 2, 3, 3, and 3, CN_{i+1} is (2,3,3,3).

When VE_i and HE_i are generated from the left column to the right column of a grid in this enumeration method, these CNs are also propagated from the left column to the right column by the generated VE_i and HE_i. Here, we show how CNs are propagated to the next column by VE_i and HE_i in detail for our new enumeration process.

When our column-sweeping method starts, the process starts from the leftmost column, column 0, and CNs are initially assigned to CN_0. As shown in Figure 8.8(a), in an MxN grid, CN_0 initially have CNs from 0 to N-1. When VE_0 and HE_0 are generated on column

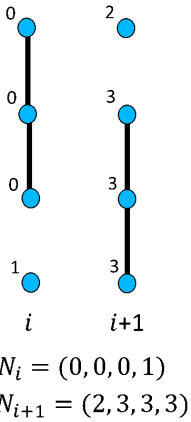


Figure 8.7: Example of nodes with CNs for column i and column i+1.

0, VE_0 is applied first. Then, the initial CN_0 is changed by VE_0 as shown in Figures 8.8(b) and 8.8(c). Given that VE_0 connects nodes on column 0 vertically, these nodes become connected and have the same CN. When the new CN is applied to the connected nodes, they are given the minimum CN between their original CNs. Please note that, when each edge on VE_0 is placed on column 0 sequentially, CNs are propagated sequentially, as shown in Figures 8b and 8c. After VE_0 is applied, HE_0 is applied. Since HE_0 connects nodes on column 0 to nodes on column 1 horizontally, the CNs in CN_0 are propagated to the next column as shown in Figure 8.8(d). The nodes on column 1 connected to column 0 will have the same CN. If some nodes on column 1 are not connected to column 0 by horizontal edges, the nodes will have CNs, which are different from the CNs of other connected nodes on column 1 as shown in Figure 8.8(e). The CNs of the unconnected nodes are different from each other because the nodes are not connected. Thus, CN_1 can show a connectivity status of nodes on column 1 correctly as shown in Figure 8.8(e), and when VE_1 and HE_1 are generated and applied on column 1, the CN_1 is changed and propagated to the next

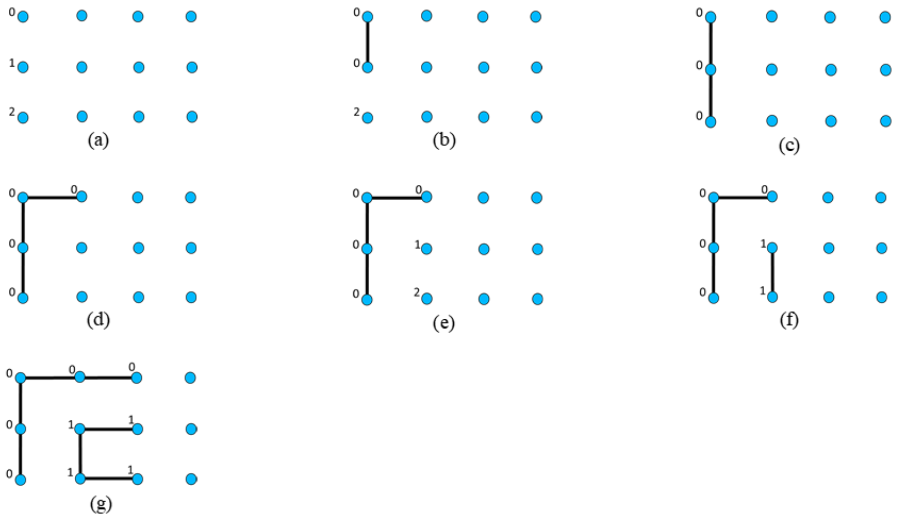


Figure 8.8: Illustration of the propagation of CNs over columns by VE_i and HE_i. (a) CN_0 is initialized as (0,1,2). (b), (c) VE_0 = (1,1) is applied on column 0, and CN_0 is changed as (0,0,0) (d) HE_0 = (1,0,0) is applied on column 0. (e) New CNs are assigned to nodes not connected to column 0 by HE_0. They have CNs, which are different from the CN of the first node on column 1 and also different from each other. A new CN_1 = (0,1,2) is obtained, (f) VE_1 = (0,1) is applied, and CN_1 is changed as (0,1,1). (g) HE_1 = (1,1,1) is applied on column 1, and a new CN_2 = (0,1,1) is obtained.

column 2 as shown in Figures 8.8(f) and 8.8(g). This CN_i propagation process continues with VE_is and HE_is until it reaches the last column.

When VE_i and HE_i are generated on column i, because not all VE_i and HE_i are valid, we need rules to generate valid VE_i and HE_i, which do not yield loops or isolated components. In the next section, we introduce these rules, using only CN_i, to generate VE_i and HE_i so as not to have loops and isolated components between the first column and the current column.

8.4.2 Rules for valid VE_i and HE_i generation with CN_i

Note that CN_i shows the connectivity status of the corresponding nodes on the column. For example, when we have CN_i =(0,1,2,0), since the first node and the last node on column i have the same CN in CN_i, we can know that they are connected on a grid between the first column and the current column. Thus we only have to examine the previous column in our enumeration. Using CN_i. we define rules to prevent loops and isolated components, without investigating any other columns.

Rule 1: VE_i should not connect nodes with the same CNs in CN_i.

When nodes on CN_i have the same CN, they are in the same connected components. In this case, when vertical edges in VE_i connect nodes with the same CN on column i, a loop appears on a grid. By generating VE_i satisfying Rule 1 on column i, we can prevent loops between the first column and column i. Note that only VE_i needs to deal with this rule.

Rule 2: At least one horizontal edge on HE_i should be connected to each connected component on CN_i.

As shown in Figure 8.9, going from the left column to the right column of a grid, when the connected component is not propagated (connected) to the next column by HE_i, the component becomes an isolated component. Thus, each connected component on CN_i should be connected to the next column by horizontal edges so as not to be an isolated component. Since there is always at least one component on CN_i, we generate HE_i with at least one 1 bit by Rule 2, so HE_i satisfies the constraint of HBV, $|\mathrm{HE}_i| \geq 1$. By generating HE_i satisfying Rule 2 on column i, we avoid isolated components between the first column and the current column i.

Rule 3: CN_{M-1} has only 1 component.

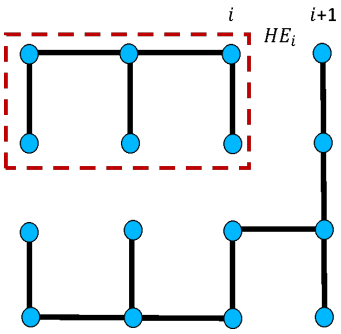


Figure 8.9: Example of having an isolated component by violating Rule 2. Since there is no horizontal edge on HE_i, which is connected to the component with red-dashed box, the component becomes an isolated component.

Together, Rule 1 and Rule 2 are a necessary and sufficient condition for having a spanning tree over a grid except on the last column.

Rule 1 ensures VE_i does not generate loops between the first column and column i, and Rule 2 ensures HE_i does not generate isolated components between the first column and column i. When VE_i and HE_i satisfy Rules 1 and 2, loops and isolated components are avoided.

However, when it comes to the last column M-1, the two rules are not sufficient to have a spanning tree. VE_{M-1} satisfying Rule 1 is generated on the last column. On the last column, when there are nodes with different CNs on CN_{M-1}, such as CN_{M-1} = (0,0,1,2), we have disconnected components. Rule 1 is not sufficient to connect different components on the last column; so when a VE_{M-1} satisfying Rule 1 is generated, we need to check if the VE_{M-1} connects all different connected components into the same connected component on the last column. For example, when we have CN_{M-1} = (0,0,1,2), VE_{M-1} = (0,1,1) should be applied to have a spanning tree.

We only need to generate VE_i and HE_i satisfying Rule 1 and Rule 2 without any checking process until it comes to the last column. Only on the last column M-1, we investigate whether the generated VE_{M-1} satisfies Rule 3 to have a spanning tree.

8.4.3 Enumeration of VE_i and HE_i on column i with Rules

Now, we explain how to enumerate VE_i and HE_i satisfying Rule 1 and Rule 2 on column i. The simple way to enumerate VE_i and HE_i is to generate all possible bit patterns on VE_i and HE_i; and for each pair of VE_i and HE_i, we check whether the pair satisfies the rules.

The pseudocode for using this simple algorithm is located in Algorithm 4. In the pseudocode, VEiSet contains all possible bit patterns for VE_i. HEiSet contains all possible bit patterns except the bit pattern with 0 bits, (0,0,0,...,0), for HE_i. When the process checks if the generated VE_i satisfies Rule 1, it iterates over each vertical edge ve on VE_i and checks whether adjacent nodes of ve have the same CN. If ve violates Rule 1, the process applies another VE_i. When ve does not violate Rule 1, using the CNiUpdatedByVe() function, the input CN_i is updated by ve as shown in Figure 8.8. After iterating over all ves without violation, VE_i is determined as valid one. Next, CN_i is assigned to $newCN_i$. Then, the process checks if HE_i from HEiSet satisfies Rule 2 with the new CN_i. In this checking process, it iterates over horizontal edges on HE_i and checks if every connected component on the new CN_i has at least one horizontal edge. If HE_i satisfies Rule 2, the pair of VE_i and HE_i is returned.

Here, we analyze the running time of Algorithm 4. In Algorithm 4, VEiSet contains 2^{N-1} bit patterns, and HEiSet contains $2^N - 1$ bit patterns. Thus, the running times for iterating VE_is and HE_is are O(2^{N-1}) and O($2^N - 1$), respectively. For each VE_i, it iterates over each edge of the VE_i to check whether VE_i satisfies Rule 1. Thus, the running time

Algorithm 4 Simple way to enumerate VE_i and HE_i based on CN_i

```
1: function SimpleWayToEnumVEiHEi(CN_i)
2:     validVEi ← TRUE
3:     validHEi ← TRUE
4:     for each VE_i ∈ VEiSet do
5:         validVEi ← TRUE
6:         for each ve ∈ VE_i do
7:             if ve violates Rule 1 with CN_i then
8:                 validVEi ← FALSE
9:                 BREAK
10:            else
11:                CN_i ← CNiUpdatedByVe(ve)
12:            end if
13:        end for
14:        if validVEi then
15:            newCN_i ← CN_i
16:            for each HE_i ∈ HEiSet do
17:                validHEi ← TRUE
18:                for each he ∈ HE_i do
19:                    if he violates Rule 2 with newCN_i then
20:                        validHEi ← FALSE
21:                        BREAK
22:                    end if
23:                end for
24:                if validHEi then
25:                    Return (VE_i, HE_i)
26:                end if
27:            end for
28:        end if
29:    end for
30: end function
```

for the checking process for each VE_i is O($N-1$). Likewise, for each HE_i, it iterates over each edge of HE_i to check whether HE_i satisfies Rule 2, so the running time for the checking process for each HE_i is O(N). The total running time for this pseudo code will be O($2^{N-1} * (N + (2^N - 1) * (N - 1))$), which can be reduced to O($2^{N-1} * N + 2^{N-1} * 2^N * N$). O($^{2N-1} * N + 2^{N-1} * 2^N * N$) can be reduced to O($2^{2*N} * N$) = $O(4^N * N)$.

We cannot say that this method is an efficient way to enumerate VE_i and HE_i satisfying Rule 1 and Rule 2 on each column. In this method, since we generate all possible bit patterns for VE_i and HE_i, it has unnecessary bit patterns based on CN_i. For example, when CN_i = (0,0,0,0), since all nodes in CN_i include the same CN, when VE_i is generated, only VE_i with 0 bits, VE_i = (0,0,0), is allowed. Other VE_i will generate a loop. Thus, if we use Algorithm 4 to generate valid VE_i in this case, we generate $2^{N-1} - 1$ unnecessary bit patterns. We perform the checking process on every bit pattern, which is time-consuming like the scanning process of the previous column-sweeping method. This simple VE_i and HE_i generation method can make the intelligent column-sweeping method even less efficient than the brute-force enumeration method with VBV and HBV.

8.5 An Intelligent Column-Sweeping Method with Lookup Table

In order to generate valid VE_i and HE_i efficiently, we can pre-generate all valid VE_i and HE_i satisfying Rule 1 and Rule 2 based on CN_i and store them in a separate place such as a lookup table, from which entries can be accessed quickly. Then, when CN_i is provided during the enumeration process, we retrieve only valid VE_is and HE_is rapidly from the table without any extra computation. In this section, we introduce a lookup table and show how we optimizes the intelligent column-sweeping method.

CN_i	VE_i
(0,0,0)	(0,0)
(0,0,1)	(0,0), (0,1)
(0,1,0)	(0,0), (0,1), (1,0)
(0,1,1)	(0,0), (1,0)
(0,1,2)	(0,0), (0,1), (1,0), (1,1)

Table 8.1: Lookup table, in which CN_i is the input and a set of VE_is is the output.

new CN_i	HE_i
(0,0,0)	(0,0,1), (0,1,0), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)
(0,0,1)	(0,1,1), (1,0,1), (1,1,1)
(0,1,0)	(0,1,1), (1,0,1), (1,1,0), (1,1,1)
(0,1,1)	(1,0,1), (1,1,0), (1,1,1)
(0,1,2)	(1,1,1)

Table 8.2: Lookup table, in which a new CN_i is the input and a set of HE_is is the output.

The purpose of a lookup table is to save a time computing valid VE_is and HE_is given the current state of CN_is. We create a one-dimensional lookup table, where the input is CN_i and the output is pairs of valid VE_i and HE_is. However, with pairs of VE_i and HE_is as entries, we may need to store a large combination of VE_i and HE_is. The VE_i only depends on CN_i. However, HE_i depends on an updated CN_i after VE_i is applied. We suggest two separate lookup tables. For each VE_i, we can store all valid HE_i as its pair. Thus, the table needs $O(2^{N-1} * 2^N)$ spaces for each CN_i. If we divide a table in a way that one table contains VE_i and another table contains HE_i, since two tables need $O(2^N) + O(2^{N-1})$ spaces, we can save space by using two lookup tables rather than one lookup table. Thus, in our research, we use two lookup tables, and examples of the tables are shown in Tables 8.1 and 8.2.

CN_i	**{**VE_i**, new**CN_i**}**
(0,0,0)	{(0,0), (0,0,0)}
(0,0,1)	{(0,0), (0,0,1)}, {(0,1), (0,0,0)}
(0,1,0)	{(0,0), (0, 1, 0)}, {(0,1), (0,0,0)}, {(1,0), (0,0,0)}
(0,1,1)	{(0,0), (0,1,1)}, {(1,0),(0,0,0)}
(0,1,2)	{(0,0), (0,1,2)}, {(0,1),(0,1,1)}, {(1,0), (0,0,1)}, {(1,1),(0,0,0)}

Table 8.3: Lookup table F, in which CN_i is the input, and pairs of VE_i and new CN_i are the output.

We obtain valid bit patterns from the lookup table in Table 8.1 based on the current CN_i, next, calculate a new CN_i based on the retrieved bit patterns for VE_i. For example, when we have a valid VE_i based on initial CN_i from Table 8.1, we compute a new CN_i, in which CNs are changed by VE_i. Then, using the changed CN_i, we obtain a valid HE_i from Table 8.2 and calculate the new CN_{i+1} based on HE_i.

When we compute a new CN_i from VE_i or a new CN_{i+1} from HE_i, the process iterates over the column, and a running time for this computation can be at least O(N). This process also requires any CN_i values to be normalized. For example, when we have three components on the column i, CN_i can be (0,1,2), (2,1,0), or (1,2,0). But, in table entry, CN_i should be normalized to only (0,1,2). If we do not want to spend time O(N) in the enumeration process for computing new CN_i and CN_{i+1}, we can store those new CN_i and CN_{i+1} in lookup tables and retrieve them rapidly instead of computing them from VE_i and HE_i. Using lookup tables can save time, but storing CN_is in lookup table requires extra space for tables. In our research, since we want to save time rather than space in the enumeration process, we construct lookup tables, which contain computed new CN_is and CN_{i+1}s as shown in Tables 8.3 and 8.4.

new CN_i	**{**HE_i, CN_{i+1}**}**
(0,0,0)	{(0,0,1),(0,0,1)}, {(0,1,0),(0,1,0)}, {(1,0,0),(0,1,1)}, {(0,1,1),(0,1,1)}, {(1,0,1),(0,1,0)}, {(1,1,0),(0,0,1)}, {(1,1,1),(0,0,0)}
(0,0,1)	{(0,1,1),(0,1,2)}, {(1,0,1),(0,1,2)}, {(1,1,1),(0,0,1)}
(0,1,0)	{(0,1,1),(0,1,2)}, {(1,0,1),(0,1,0)}, {(1,1,0),(0,1,2)}, {(1,1,1),(0,1,0)}
(0,1,1)	{(1,0,1),(0,1,2)}, {(1,1,0),(0,1,2)}, {(1,1,1),(0,1,1)}
(0,1,2)	{(1,1,1),(0,1,2)}

Table 8.4: Lookup table G, in which new CN_i is the input and pairs of HE_i and CN_{i+1} are the output.

Tables 8.3 and 8.4 show examples of lookup tables F and G that we use in the intelligent column-sweeping method. In table F, input entries are normalized CN_i, and output entries are tuples of valid VE_i and new normalized CN_is. In table G, input entries are new CN_is from Table F, and output entries are tuples of valid HE_is and new normalized CN_{i+1}s.

The pseudocode for enumerating valid VE_i and HE_i with lookup tables is shown in Algorithm 5. In Algorithm 5, with given CN_i, F(CN_i) retrieves all possible tuples of valid VE_i and new CN_is from table F and stores them to VEiCNiSet. For each pair of VE_i and new CN_i, G(newCN_i) retrieves all possible tuples of valid HE_i and CN_{i+1} from table G. A tuple of valid VE_i, valid HE_i and CN_{i+1} is returned. CN_{i+1} will be used as input to table F when the process retrieves valid VE_{i+1} and new CN_{i+1} on the next column i+1, and this process of using the CN_{i+1} for the next column i+1 will be described when entire pseudocode of the intelligent column-sweeping method is explained later by Algorithm 5.

Now, we analyze the running time for Algorithm 5. In table F, since new CN_i is dependent on VE_i, the number of pairs of valid VE_i and new CN_i equals to the number of valid VE_i, and the number of valid VE_i is less than 2^{N-1} for some input CN_is. Since the number of VE_i is different for each CN_i input in table F, we use the average number of pairs of valid VE_i and new CN_i, which we call f in the analysis of running time. Likewise,

Algorithm 5 Way to enumerate VE_i and HE_i based on CN_i using lookup tables F and G

```
1: function EnumVEiHEiwithLookupTable(CN_i)
2:     VEiCNiSet ← F(CN_i)
3:     for each VEiCNi ∈ VEiCNiSet do
4:         VE_i ← VEiCNi.VE_i
5:         newCN_i ← VEiCNi.CN_i
6:         HEiandCNiplus1Set ← G(newCN_i)
7:         for each HEiCNiplus1 ∈ HEiCNiplus1Set do
8:             HE_i ← HEiCNiplus1.HE_i
9:             CN_{i+1} ← HEiCNiplus1.CN_{i+1}
10:            Yield (VE_i, HE_i, CN_{i+1})
11:        end for
12:    end for
13: end function
```

in table G, we use the average number of pairs of valid HE_i and new CN_{i+1}, which we call g in the analysis of running time. Since there is no computation required to enumerate valid VE_i and HE_i based on CN_i on column i, the running time for Algorithm 5 is O(f*g), where f $< 2^{N-1}$ and g $< 2^N$. We can see that by using lookup tables, the running time for enumerating valid VE_i and HE_i is reduced more than O(N) times from the running time of the previous simple method (Algorithm 4), which is O($4^N * N$). Furthermore, each valid spanning tree is produced in O(1) time. Thus, the use of lookup tables can save computation time.

The pseudocode for the intelligent column-sweeping method with lookup tables is shown in Algorithm 6. It starts with IntelColSweep(0, ∅, ∅, (0,1,2,...,N-1)). On each recursion, VE_i and HE_i satisfying Rules 1 and 2, and CN_{i+1} are retrieved from tables F and G. CN_{i+1} is used in the next recursion, in which valid VE_{i+1} and HE_{i+1} are generated on the next column i+1. When it comes to the last column M-1, as discussed previously, the process checks if VE_{M-1} connects all nodes with different CNs on CN_{M-1}. The process finds VE_{M-1} which has new CN_{M-1}, in which all nodes have the same CN like (0,0,0,0,...,0) in

table F. When VE_{M-1} with new CN_{M-1}, in which all nodes have the same CN, is found, VBV and HBV are returned as a spanning tree.

Algorithm 6 Intelligent Column-Sweeping Method with Lookup Tables

```
1: function IntelColSweep(i, VBV, HBV, CN_i)
2:     if i = M − 1 then
3:         VEiCNiSet ← F(CN_i)
4:         for each VEiCNi ∈ VEiCNiSet do
5:             VE_i ← VEiCNi.VE_i
6:             newCN_i ← VEiCNi.CN_i
7:             if All nodes in newCN_i contain the same CN then
8:                 Return (VBV ∪ VE_i, HBV)
9:             end if
10:        end for
11:    else
12:        VEiCNiSet ← F(CN_i)
13:        for each VEiCNi ∈ VEiCNiSet do
14:            VE_i ← VEiCNi.VE_i
15:            newCN_i ← VEiCNi.CN_i
16:            HEiandCNiplus1Set ← G(newCN_i)
17:            for each HEiCNiplus1 ∈ HEiCNiplus1Set do
18:                HE_i ← HEiCNiplus1.HE_i
19:                CN_{i+1} ← HEiCNiplus1.CN_{i+1}
20:                IntelColSweep (VBV ∪ VE_i, HBV ∪ HE_i, CN_{i+1})
21:            end for
22:        end for
23:    end if
24: end function
```

Now, we analyze the running time for Algorithm 6. On each column, the running time for generating VE_i and HE_i with lookup tables is O(f*g). When the process is on the last column, it can investigate whether nodes in CN_{M-1} have the same CN by iterating over all CNs on CN_{M-1}. Since the checking process is performed for each VE_{M-1}, the running time on the last column can be O(f*N) in this case. This running time can be optimized more if we use a separate lookup table for the last column, where valid VE_{M-1}, which has desired CN_{M-1} = (0,0,0,...,0) as its pair in lookup table F, is only stored as shown

CN_{M-1}	VE_{M-1}
(0,0,0)	(0,0)
(0,0,1)	(0,1)
(0,1,0)	(0,1), (1,0)
(0,1,1)	(1,0)
(0,1,2)	(1,1)

Table 8.5: Separate lookup table for obtaining a valid VE_{M-1} on the last column.

in Table 8.5. Thus, if we use Table 8.5 on the last column, it takes O(1) to obtain valid VE_{M-1} to have a spanning tree based on CN_{M-1}. Then, the running time of the process on the last column is optimized as O(f) because no checking process is performed. Since this column-based enumeration process goes from the left column to the right column in a recursive manner, and there are total M columns on a grid including the last column, the total running time of this intelligent column-sweeping method is O((f*g)$^{M-1}$*f), where f $< 2^{N-1}$ and g $< 2^N$. The running time can be reduced to O((f*g)^M).

In this intelligent column-sweeping method sweeping from the left column to the right column of a grid, the enumeration process does not need any checking process except on the last column. Even this checking process can be removed if a separate lookup table, such as Table 8.5, is used. It does not generate duplicate spanning trees, invalid VBVs and invalid HBVs. Using lookup tables, we optimized the enumeration process so that it has a running time of O((f*g)^M) to find all spanning trees. This running time is in fact optimal as each spanning tree is computed in O(N*M) if bit operations are included. The running time is much lower than O($(M * N)! * M * N * \log(M * N)$), which is a running time of brute-force enumeration method using an MST algorithm. Moreover, the running time is lower than the running time of the brute-force enumeration method using VBV and HBV (O($4^{M*N} * M * N$)).

8.6 Estimating Size of Lookup Tables

When we use lookup tables, we may need to consider whether they are too large to store in memory. Since our enumeration process is performed in a small grid ($\leq$ 6x6), in this section, we estimate size of the look tables and check whether they have a reasonable size to store in a 6x6 grid.

To explain the table size estimation easier, we define the operator

for CN_i. When we call $CN_i[0]$, it represents the CN of the top-most node in the CN_i. Likewise, when we call $CN_i[2]$, it represents CN of the third node from the top of the CN_i. For example, for CN_i=(0,1,2), $CN_i[0]$ is 0 and $CN_i[2]$ is 2.

As mentioned in the previous section, CN_i is normalized in the table entry. Each node of CN_i has different range as shown in below.

$$CN\,[0] = 0$$
$$0 \leq CN[1] \leq 1$$
$$0 \leq CN[2] \leq 2$$
$$0 \leq CN[3] \leq 3$$
$$\vdots$$

Thus, we can calculate the number of all possible CN_is without duplicates as $\mathrm{O}(1 * 2 * 3 * * N) = \mathrm{O}(N!)$, which indicates that both lookup tables F and G have $\mathrm{O}(N!)$ entries. In a 6x6 grid, each CN of a CN_i is not bigger than 6, so we need 3 bits to represent each CN. Because there are a total 6 CNs in CN_i, each CN_i needs $6 * 3 = 18$ bits to be represented. It denotes that each lookup table needs $\mathrm{O}(6! * 18)$ bits to store all CN_is for a 6x6 grid.

Let's estimate a size of output entries of the table F. Table F has pairs of VE_i and new CN_i as output entries. Since there are at most $O(2^{N-1})$ bit patterns on VE_i, there are at most $O(2^5)$ output entries for each input of table F in a 6x6 grid. In a 6x6 grid, we need 5 bits for VE_i and 18 bits for new CN_i. Thus, for each input CN_i, table F needs at most $O(2^5 * (5 + 18))$ bits to store corresponding output entries.

Now, let's estimate a size of output entries of the table G. Table G has pairs of HE_i and CN_{i+1} as output entries. Since HE_i has N bits, we need 6 bits for HE_i in a 6x6 grid. Also, we need 18 bits for CN_{i+1} like new CN_i in table F. There are at most $O(2^N - 1)$ bit patterns on HE_i, so, in a 6x6 grid, there are at most $O(2^6 - 1)$ output entries for each CN_i. Thus, for each input CN_i, table G needs at most $O((2^6 - 1) * (6 + 18))$ bits to store output entries.

Therefore, the total size of lookup tables in a 6x6 grid is $O(6! * 18 * (2^5 * (5 + 18) + (2^6 - 1) * (6 + 18))) = O(29134080)$ bits, which is O(3.64) Megabyte. Since this is upper bound of size for lookup tables, the actual size of the tables will be less than O(3.64) Megabyte. Even with O(3.64) Megabyte, we can say that lookup tables have an affordable size to store and use during the enumeration process.

In this chapter, we explained our spanning tree enumeration algorithm using VBV and HBV. However, in our research, we consider spanning trees with symmetric topologies as duplicate spanning trees and do not want the enumerated space to have them. In Figure 8.10, spanning trees have symmetric topologies in a grid, and they are considered as duplicate spanning trees. In the next chapter, we define duplicate spanning trees in more detail and, based on our enumeration algorithm, show how to obtain the enumerated space of spanning trees without these duplicates.

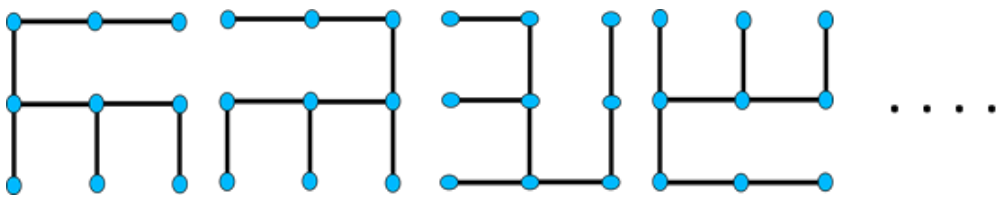


Figure 8.10: Spanning trees with symmetric topologies in a 3x3 grid. These spanning trees are considered as duplicates in our research.

Chapter 9: Distinct Spanning Tree Enumeration with Edge Bit Vectors

In Chapter 8, we introduced our spanning tree enumeration algorithm. Using the algorithm, we can obtain an enumerated set of spanning trees. And, with this enumerated space, we can determine the characteristics of the existing maze generation algorithms. However, in the enumerated space, we wish to remove or avoid spanning trees with with symmetric topologies. To explain how we enumerate non-symmetric spanning trees, we first show what kinds of symmetric spanning trees can be considered as duplicated ones on a grid first. Note that the flow of enumeration process is a little bit different compared to the previous chapter. In the previous chapter, VBV and HBV enumeration were interleaved in the enumeration process. Instead, in this chapter, VBVs will be enumerated first, and valid HBVs, which make spanning trees with the VBVs, will be enumerated next because this approach can make the enumeration process efficient by filtering VBVs in advance. The filtering process will be discussed in Section 9.5.1.

9.1 Symmetric Spanning Trees

In a regular grid, our research defines seven types of symmetries, which can be divided into mirrored and rotated symmetric types. In mirrored symmetric types, there are 1) vertically mirrored symmetry, 2) horizontally mirrored symmetry, and 3) both vertically and horizontally mirrored symmetry. In rotationally symmetric types, there are 4) rotated

symmetry, 5) rotated with vertically mirrored symmetry, 6) rotated with horizontally mirrored symmetry, and 7) rotated with both vertically and horizontally mirrored symmetry. As examples, Figure 9.1 depicts a spanning tree and its seven symmetric spanning trees. Spanning trees in Figure 9.1(b), Figure 9.1(c), and Figure 9.1(d) are ones mirrored vertically, mirrored horizontally and mirrored in both dimensions, respectively. Spanning tree in Figure 9.1(e) is rotated in 90-degrees in a clockwise direction. From now on, the expression rotated in 90-degrees in a clockwise direction will be abbreviated to rotated. Spanning trees in Figure 9.1(f), Figure 9.1(g), and Figure 9.1(h) are spanning trees rotated and mirrored vertically, rotated and mirrored horizontally, and rotated and mirrored in both dimensions, respectively. These seven symmetric spanning trees are considered duplicates of the spanning tree in Figure 9.1(a) as they have the same basic as they have the same basic topology and metrics.

Based on the number of columns and the number of rows of a grid, symmetric spanning trees can have less symmetric types. In an MxN grid, where $M = N$, all seven symmetric types can exist for symmetric spanning trees. In an MxN grid, where $M \neq N$, since a spanning tree cannot rotate on a grid, only mirrored symmetric types like ones in Figure 9.1(b), Figure 9.1(c) and Figure 9.1(d), can exist for symmetric spanning trees.

In our research, each spanning tree has its set of symmetric spanning trees, which denotes the set that contains itself and its all symmetric spanning trees. For example, for the spanning tree in Figure 9.1(a), its set of symmetric spanning trees is the set which contains all eight spanning trees in Figure 9.1. Likewise, for the spanning tree in Figure 9.1(b), its set of symmetric spanning trees is also a set of all spanning trees in Figure 9.1. Thus, if a spanning tree α is symmetric to a spanning tree β, their sets of symmetric

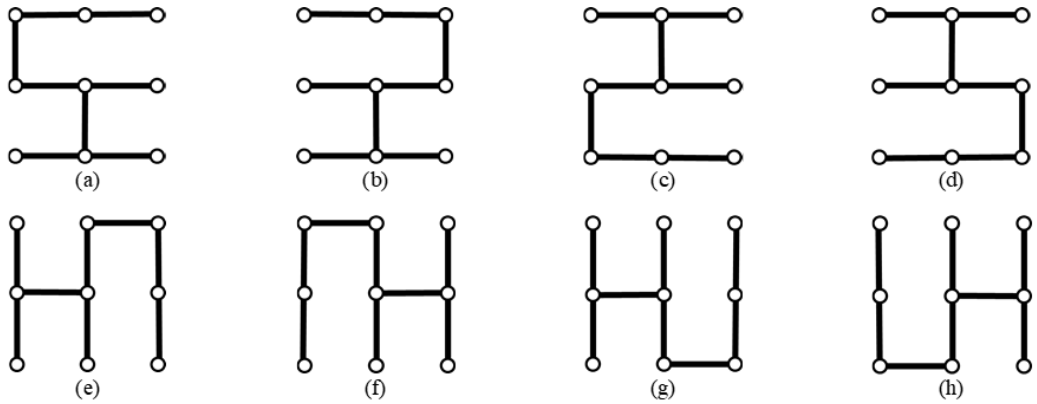


Figure 9.1: Figures to show a spanning tree (a) and its seven symmetric spanning trees in a grid: (b) mirrored horizontally; (c) mirrored vertically; (d) mirrored both vertically and horizontally; (e) rotated in 90-degree clockwise direction; (f) rotated and mirrored vertically; (g) rotated and mirrored horizontally; and (h) rotated and mirrored both vertically and horizontally.

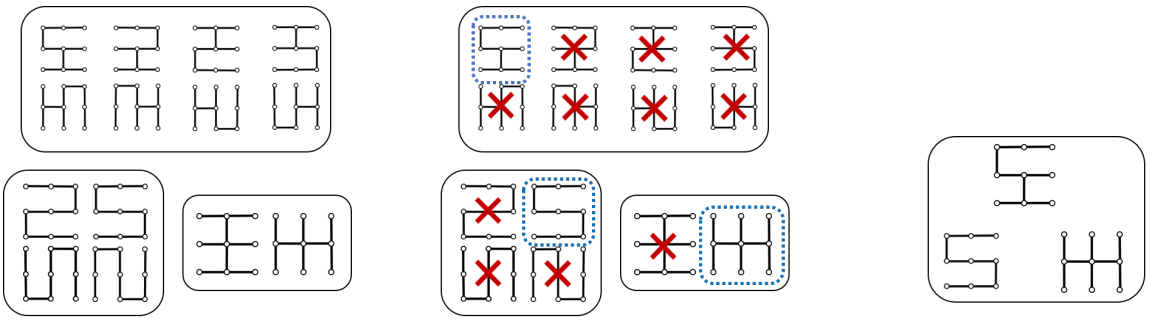
spanning trees are equivalent. Note that in this chapter, we will call the set as a symmetric group.

9.2 Distinct Spanning Trees

In a set of spanning trees, when spanning trees are not symmetric to each other, those spanning trees are called distinct spanning trees.

Here is a basic idea of how we obtain a set of distinct spanning trees. As shown in Figure 9.2, when we have a set of spanning trees with several symmetric groups, we simply remain one spanning tree from each symmetric group and remove rest of them. Then, the set of remained spanning trees will be a set of distinct spanning trees.

In the following sections, we provide ways to obtain an enumerated set of all possible distinct spanning trees.



(a) Symmetric Groups (b) Remaining one spanning tree in each symmetric group (c) Distinct Spanning Tree Set

Figure 9.2: An overview of having a set of distinct spanning trees from symmetric groups.

9.3 Brute-Force Method of Distinct Spanning Tree Enumeration

In order to obtain the enumerated set of distinct spanning trees, first we enumerate and store all possible spanning trees. Then, we cluster the enumerated set based on symmetric groups. After clustering the enumerated set, we remain one spanning tree from each symmetric group, and the set of remained spanning trees are an enumerated set of distinct spanning trees.

This method yields the enumerated set, but we need to enumerate all possible spanning trees and also cluster them by symmetric groups. Because even a small grid, such as 6x6, has trillions of all possible spanning trees, enumerating and clustering the spanning trees will not be feasible.

9.4 Efficient Method of Distinct Spanning Tree Enumeration

In this section, we provide an efficient method, which does not need to store and cluster all possible spanning trees.

To avoid storing and clustering all possible spanning trees, we need to redefine distinct spanning trees. In Section 9.2, a distinct spanning tree was defined as a remained spanning

tree in a symmetric group, and it can be any spanning tree of the symmetric group. But now we define a distinct spanning tree as one designated spanning tree in a symmetric group. It indicates that in each symmetric group, the same spanning tree is always returned as a distinct spanning tree.

The next question is which spanning tree is designated as a distinct spanning tree in a symmetric group. As explained in Chapter 6, a spanning tree is represented by two bit-vectors VBV and HBV. Since the bit-vectors are binary representation, a spanning tree can be converted to two integers. There can be various ways to determine distinct spanning tree from each symmetric group, but in our research, we use the integers converted from spanning tree in the designation process.

First, we will explain about distinct spanning tree designation with the integers with more details. Next, we will explain the efficient enumeration process using the integers.

9.4.1 Distinct Spanning Tree Designation with Integers of Spanning Trees

Since a spanning tree is represented by two binary bit-vectors, VBV and HBV, the tree can be converted to two integers as shown in Figure 9.3. On each bit vector, the integer computation goes from the right to the left. For example, when VBV is (1,0,0,1,1,0) in a 3x3 grid, its corresponding integer is $0 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 0 * 2^4 + 1 * 2^5 = 38$.

Then, when we have spanning trees in a symmetric group, the spanning tree, which has two maximum integers amongst, is designated as a distinct spanning tree. However, we have observed that there is a case in which no spanning tree has the maximum integer of VBV and the maximum integer of HBV at the same time. In Figure 9.4, we can see that there is no spanning tree which has maximum integers of both VBV and HBV. In this case, we compare integer of VBV ($VBVInt$) to $VBVInt$s first. The spanning tree with the maximum $VBVInt$ is considered as a distinct spanning tree. If they have the same

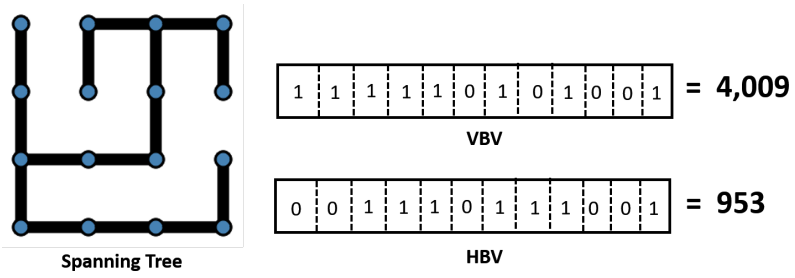


Figure 9.3: Example of spanning tree and its corresponding two integers. Each integer is obtained from the corresponding EBV.

maximum $VBVInt$, we compare their integers of HBV ($HBVInt$), and a spanning tree with the maximum $HBVInt$ is designated as a distinct spanning tree. We name the spanning tree that has the maximum integers in this manner as distinct spanning tree. For example, in Figure 9.4, when we investigate which spanning tree has the maximum $VBVInt$, we can see that all four spanning trees in the first line have the maximum $VBVInt$, which is 63. Then, we check which spanning tree has the maximum $HBVInt$ among them. The spanning tree that has $HBVInt$=34 is designated as a distinct spanning tree.

9.4.2 Enumeration Process

While we enumerate all possible spanning trees, for each spanning tree, we retrieve its symmetric spanning trees. Then, we check whether the spanning tree has the maximum integers compared to its symmetric spanning trees. If the spanning tree has the maximum integers, the tree is stored as distinct spanning tree. After enumerating all possible spanning trees, we will have a set of distinct spanning trees as a result. In this process, we do not need to store all possible spanning trees in advance. Also, for each spanning tree, we only need to compare it to it symmetric spanning trees, which are at most seven spanning trees.

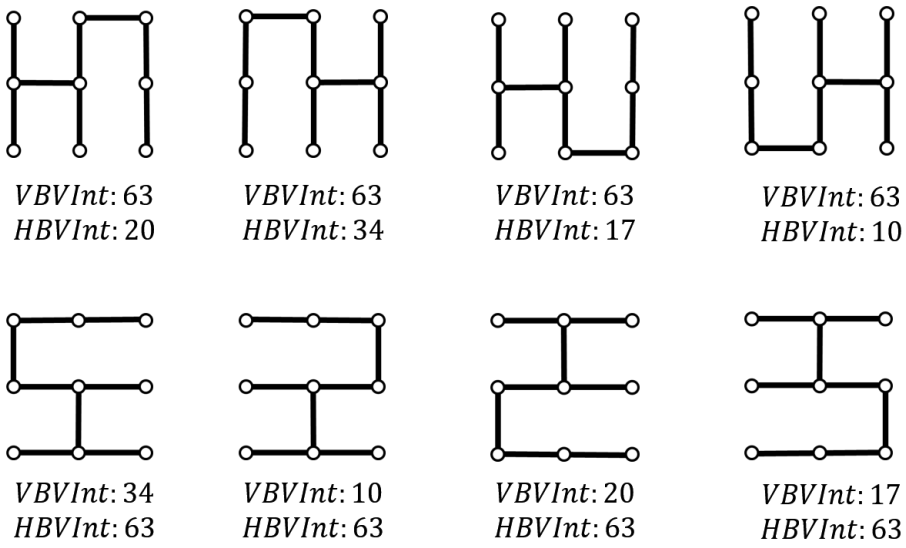


Figure 9.4: Figure that showing symmetric spanning trees with their integer of VBV ($VBVInt$) and integer of HBV ($HBVInt$). To find a distinct spanning tree amongst, we first compare $VBVInt$s to see which spanning tree the maximum $VBVInt$. Then, among spanning trees with the maximum $VBVInt$, we find the spanning tree with the maximum $HBVInt$. That spanning tree becomes a distinct spanning tree. In this figure, the spanning tree with $VBVInt$=63 and $HBVInt$=34 is designated as a distinct spanning tree.

However, we still need to generate all possible spanning trees anyhow, and it is still a time-consuming job. In the next section, we introduce a more efficient enumeration algorithm, which generates fewer spanning trees by applying some constraints.

9.5 Efficient Method of Distinct Spanning Tree Enumeration with VBV Constraints

In our enumeration process, we can filter VBVs, in advance so that we do not need to enumerate all spanning trees. In the next section, we introduce several constraints of VBV to remove symmetric spanning trees early.

9.5.1 VBV Constraints for Symmetric Spanning Trees

To explain the constraints easier, we define several sets below.

VC_1, VC_2: In an MxN grid, VC_1 denotes the number of vertical edges of a spanning tree between the first column and α $^{\text{th}}$ column of a grid, where $\alpha = \lceil \frac{M}{2} \rceil$. VC_2 denotes the number of vertical edges of a spanning tree between β^{th} column, where $\beta = \lfloor \frac{M}{2} \rfloor$, and the last column of a grid. For reference, VC_1 and VC_2 in 4x4 grid are visually represented in Figure 9.5(a). As shown in Figure 9.5, for our bit vector, the first $\left\lceil \frac{M*(N-1)}{2} \right\rceil$ bits are in VC_1, and the remainder bits are in VC_2.

VR_1, VR_2: In an MxN grid, VR_1 denotes number of vertical edges of a spanning tree between the first row and α^{th} row of a grid, where $\alpha = \lceil \frac{N}{2} \rceil$. VR_2 denotes number of vertical edges of a spanning tree between β^{th} row and the last row of a grid, where $\beta = \lfloor \frac{N}{2} \rfloor$. In other words, $VR_1 = \bigcup_{i=1}^{\alpha-1} V_i$, and $VR_2 = \bigcup_{i=\beta}^{N-1} V_i$, where V_i is a set of vertical edges between i$^{\text{th}}$ row and (i+1)$^{\text{th}}$ row as defined in Chapter 6. For reference, VR_1 and VR_2 in 4x4 grid are visually represented in Figure 9.5(b).

VCR_1, VCR_2: $VCR_1 = VC_1 \cap VR_1$ and $VCR_2 = VC_2 \cap VR_2$. For the reference, VCR_1 and VCR_2 in 4x4 grid are visually represented in Figure 9.5(c).

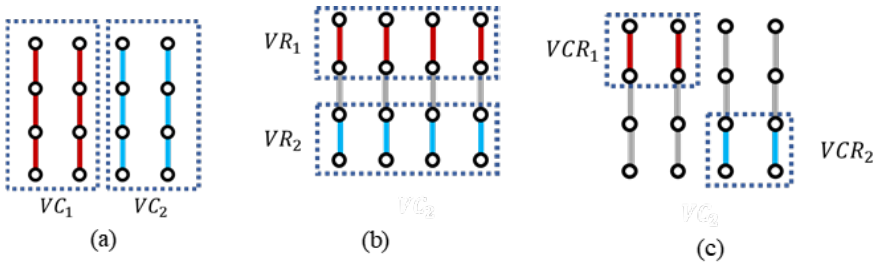


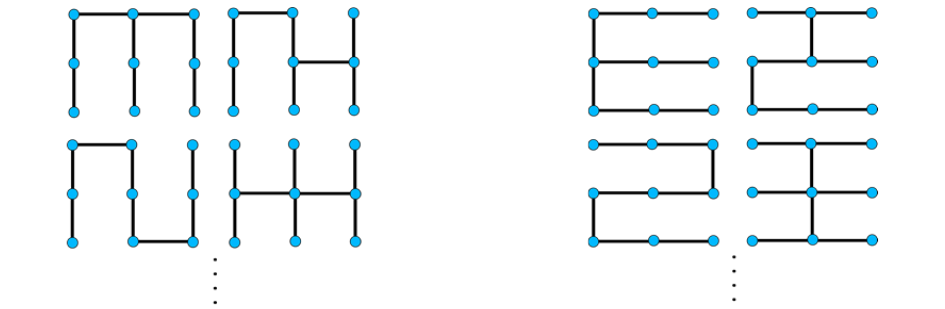
Figure 9.5: Representation of VC_1,VC_2,VR_1,VR_2,VCR_1 and VCR_2 in 4x4 grid. Sets are distinguished by different colors of vertical edges with blue-dashed boxes.

Here, we represent constraints on the VBV useful for removing symmetric spanning trees using the sets defined above.

Constraint 1. $|V| \geq \lfloor \frac{M \times N - 1}{2} \rfloor$

V represents a set of vertical edges as defined in Chapter 6. This constraint allows us to filter rotationally symmetric spanning trees. To understand this constraint easier, we explain the constraint in terms of vertical edges and horizontal edges of a spanning tree first. A set of spanning trees with $|V| \geq |H|$ is rotationally symmetric to a set of spanning trees with $|V| \leq |H|$, where H is a set of horizontal edges of spanning tree as defined in Chapter 6. Here, rotation symmetry indicates rotation by 90 degrees or 270 degrees. In a 3x3 grid, as shown in Figure 9.6, spanning trees, in which VBV has six vertical edges, and HBV has two horizontal edges, are rotationally symmetric to some spanning trees, in which VBV has two vertical edges, and HBV has six horizontal edges. By removing spanning trees with $|V| < |H|$, we can filter some rotationally symmetric spanning trees.

Since $|H| + |V| = M \times N - 1$ in an MxN grid, the constraint $|V| \geq |H|$ can be changed in terms of $|V|$ below, which is Constraint 1.



(a) Set of spanning trees where VBV has six 1 bits.

(b) Set of spanning trees where HBV has six 1 bits

Figure 9.6: Figures that show rotational symmetries between spanning trees of set (a) and set (b).

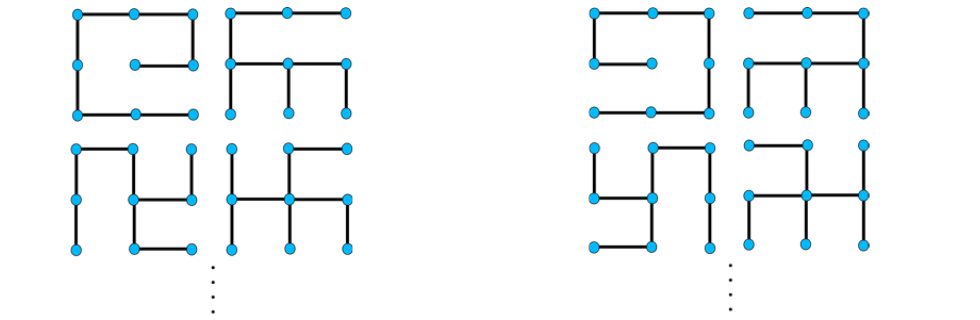
$$
\begin{aligned}
|V| \geq |H| &= M \times N - 1 - |V| \\
\Rightarrow |V| &\geq \left\lfloor \frac{M \times N - 1}{2} \right\rfloor
\end{aligned}
\tag{9.1}
$$

As we explained using the constraint $|V| \geq |H|$, spanning trees with VBVs, in which $|V| \geq \lfloor \frac{M \times N - 1}{2} \rfloor$, are rotationally symmetric to spanning trees with VBVs, in which $|V| \leq \lfloor \frac{M \times N - 1}{2} \rfloor$, so rotational symmetric spanning trees with $|V| < \lfloor \frac{M \times N - 1}{2} \rfloor$ can be filtered by generating VBVs satisfying Constraint 1.

Constraint 2. $|\mathrm{VC}_1| \geq |\mathrm{VC}_2|$

This constraint allows us to filter vertically mirrored spanning trees. A set of spanning trees with $|\mathrm{VC}_1| \geq |\mathrm{VC}_2|$ are vertically mirrored to a set of spanning trees with $|\mathrm{VC}_1| \leq |\mathrm{VC}_2|$. For example, in 3x3 grid, as shown in Figure 9.7, a set of spanning trees, in which $|\mathrm{VC}_1| = 2$ and $|\mathrm{VC}_2| = 1$, are vertically mirrored to a set of spanning trees, in which $|\mathrm{VC}_1| = 1 and |\mathrm{VC}_2| = 2$. Thus, by generating VBVs satisfying this constraint, vertically mirrored spanning trees with $|\mathrm{VC}_1| < |\mathrm{VC}_2|$ can be filtered.

Constraint 3. $|\mathrm{VR}_1| \geq |\mathrm{VR}_2|$



(a) Set of spanning trees where $|VC_1| = 2$ and $|VC_2| = 1$.

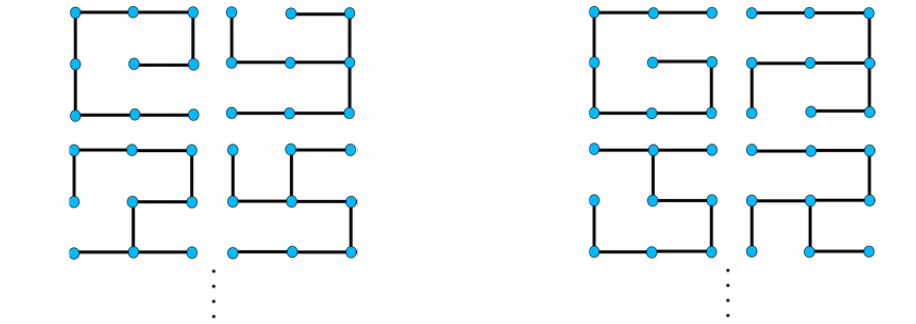
(b) Set of spanning trees where $|VC_1| = 1$ and $|VC_2| = 2$.

Figure 9.7: Figures that show vertically mirrored symmetries between spanning trees of set (a) and set (b).

This constraint allows us to filter horizontally mirrored spanning trees. A set of spanning trees with $|VR_1| \geq |VR_2|$ are horizontally mirrored to a set of spanning trees with $|VR_1| \leq |VR_2|$. For example, in 3x3 grid, as shown in Figure 9.8, a set of spanning trees, in which $|VR_1| = 2$ and $|VR_2| = 1$, are horizontally mirrored to a set of spanning trees, in which $|VR_1| = 1$ and $|VR_2| = 2$. Thus, by generating VBVs satisfying this constraint, horizontally mirrored spanning trees with $|VR_1| < |VR_2|$ can be filtered.

Constraint 4. $|VCR_1| \geq |VCR_2|$

This constraint allows us to filter symmetric spanning trees which are mirrored both vertically and horizontally. A set of spanning trees with $|VCR_1| \geq |VCR_2|$ are vertically and horizontally mirrored to a set of spanning trees with $|VCR_1| \leq |VCR_2|$. For example, in 3x3 grid, as shown in Figure 9.9, a set of spanning trees, in which $|VCR_1| = 1$ and $|VCR_2| = 0$, are vertically and horizontally mirrored to a set of spanning trees, in which $|VCR_1| = 0$ and $|VCR_2| = 1$. Thus, by generating VBVs satisfying this constraint, vertically and horizontally mirrored spanning trees with $|VCR_1| < |VCR_2|$ can be filtered. Note that



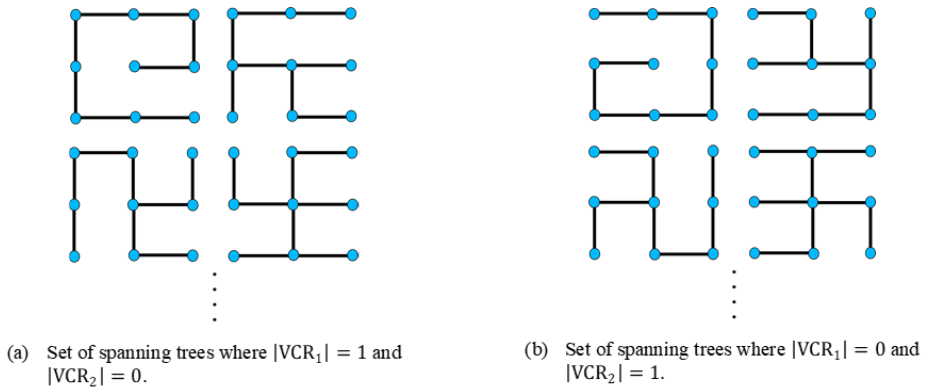
(a) Set of spanning trees where $|VR_1| = 2$ and $|VR_2| = 1$.

(b) Set of spanning trees where $|VR_1| = 1$ and $|VR_2| = 2$.

Figure 9.8: Figures that show horizontally mirrored symmetries between spanning trees of set (a) and set (b).

as shown in Figure 9.9, vertically and horizontally mirrored symmetry is the same as rotation by 180 degrees.

In our enumeration method, instead of enumerating all possible VBVs, we enumerate VBVs that satisfy these constraints. However, as shown in Figure 9.10, there is a case that the designated distinct spanning tree (spanning tree with the maximum integers of VBV and HBV in its symmetric group) does not have VBV satisfying the constraints. In this case, the distinct spanning tree will be filtered by the VBV constraints. Thus, from now on, we redefine a distinct spanning tree as a spanning tree that has the maximum integers among the symmetric spanning trees satisfying the VBV constraints. For example, in Figure 9.10, since spanning trees of Figure 9.10(c) and Figure 9.10(d) violate Constraint 1 by having $|V| < \lfloor \frac{M \times N - 1}{2} \rfloor$, distinct spanning tree is selected between spanning trees of Figure 9.10(a) and Figure 9.10(b). The spanning tree of Figure 9.10(a) is designated as a distinct spanning tree.



(a) Set of spanning trees where $|VCR_1| = 1$ and $|VCR_2| = 0$.

(b) Set of spanning trees where $|VCR_1| = 0$ and $|VCR_2| = 1$.

Figure 9.9: Figures that show symmetries, which are mirrored both vertically and horizontally between spanning trees of set (a) and set (b).

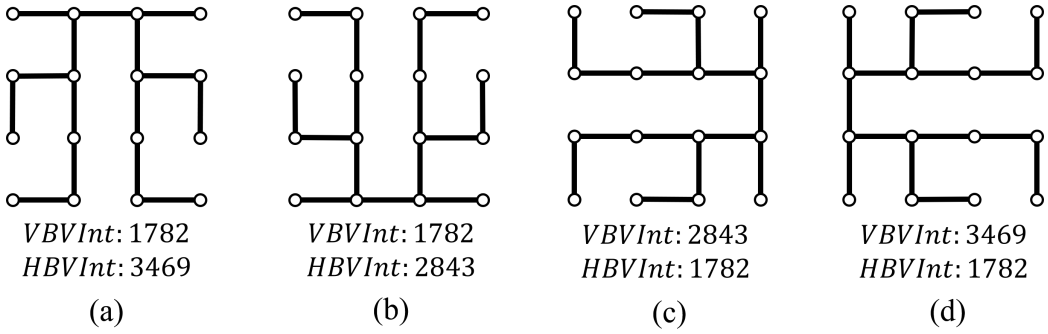


Figure 9.10: Figure showing symmetric spanning trees and their interger of VBV ($VBVInt$) and integer of HBV ($HBVInt$). In this figure, the spanning tree (d) is a distinct spanning tree in its symmetric group because it has the maximum $VBVInt$ and the maximum $HBVInt$ amongst. However, the distinct spanning tree will be filtered by Constraint 1 because it has $|V| < \lfloor \frac{M \times N - 1}{2} \rfloor$.

9.5.2 Filtering VBVs with Symmetric Checking Algorithm

The above constraints are necessary but not sufficient for removing all symmetric spanning trees. For example, when we generate VBVs with Constraint 1, in a MxN grid, where M=N and $\frac{M\times N-1}{2}$ is even value, there is a case that we have VBVs with $|V| = \frac{M\times N-1}{2}$. Since $|V| = |H|$ in this case, after we rotate a set of spanning trees with $|V| = \frac{M\times N-1}{2}$ by 90 degrees or 270 degrees, we still get a set of spanning trees with $|V| = \frac{M\times N-1}{2}$. When we generate a set of spanning trees from the VBVs, the set is same with its rotated set. It indicates that the set is rotationally symmetric to its own set. Thus, the rotationally symmetric spanning trees cannot be filtered by using only Constraint 1.

Therefore, we need an additional action to remove all possible symmetric spanning trees. We introduce symmetric checking algorithm in this section. After a VBV is generated with Constraints 1,2,3, and 4, and a HBV is generated with the VBV, the symmetric checking algorithm investigates if the pair of VBV and HBV is indeed a distinct spanning tree. If the spanning tree is confirmed as a distinct spanning tree, it is enumerated.

Algorithm 7 represents the symmetric checking algorithm. Given a VBV and a HBV, the algorithm generates all possible symmetric spanning trees. For an MxN grid, where $M = N$, all symmetries exist, and, for an MxN grid, where $M \neq N$, only mirrored symmetric spanning trees exist. Integers of symmetric spanning trees are compared with integers of the given VBV and HBV are compared. In this comparison, only symmetric spanning trees with VBVs satisfying Constraints 1,2,3, and 4 are used. The comparison is performed as described in Section 9.4.1. If the given VBV and HBV have maximum integers, the symmetric checking algorithm returns true, which denotes that the spanning tree with the VBV and HBV is a distinct spanning tree.

Algorithm 7 Symmetric chekcing algorithm in a MxN grid graph

```
1: function SymChecking(VBV, HBV)
2:     Initialize SymSTs
3:     CheckIfDistinct ← TRUE
4:     SymSTs.add(VSym(VBV, HBV))          ▷ VSym(): vertically mirrored ST.
5:     SymSTs.add(HSym(VBV, HBV))          ▷ HSym(): horizontally mirrored ST.
6:     SymSTs.add(VHSym(VBV, HBV))         ▷ VHSym(): vertically and horizontally
   mirrored ST.
7:     if M = N then
8:         SymSTs.add(RSym(VBV, HBV))      ▷ RSym(): rotated ST.
9:         SymSTs.add(RVSym(VBV, HBV))     ▷ RVSym(): rotated and vertically
   mirrored ST.
10:        SymSTs.add(RHSym(VBV, HBV))     ▷ RHSym(): rotated and horizontally
   mirrored ST.
11:        SymSTs.add(RVHSym(VBV, HBV))▷ RVHSym(): rotated and vertically and
   horizontally mirrored ST.
12:    end if
13:    maxVBVInt ← VBV.toINT
14:    maxHBVInt ← HBV.toINT
15:    for each SymST ∈ SymSTs do
16:        if SymST.VBV satisfies Constraints 1,2,3,and 4 then
17:            if maxVBVInt < SymST.VBVInt then
18:                CheckIfDistinct ← FALSE
19:            else if maxVBVInt = SymST.VBVInt then
20:                if maxHBVInt < SymST.HBVInt then
21:                    CheckIfDistinct ← FALSE
22:                end if
23:            end if
24:        end if
25:    end for
26: return CheckIfDistinct
27: end function
```

9.5.3 Distinct Spanning Tree Enumeration Algorithm with Constraints

In this section, we show the whole overview of our distinct spanning tree enumeration algorithm, which applies VBV constraints and symmetric checking. In the process, VBVs are enumerated on a grid. Valid vertical bit patterns except a bit pattern with only 0 bits are enumerated on each row. And VBVs, which do not satisfy Constraints 1,2,3, and 4, are filtered. Next, for each VBV, HBVs are enumerated. Then, the symmetric checking algorithm checks whether each pair of VBV and HBV is a distinct spanning tree or not. If the spanning tree with the VBV and HBV is a distinct spanning tree, the spanning tree is enumerated. After the symmetric checking algorithm processes all generated pairs of VBV and HBV, the set of all possible distinct spanning trees has been enumerated.

In our distinct spanning tree enumeration algorithm, for each generated spanning tree, instead of comparing it to all possible spanning trees, we only need to compare it to at most seven symmetric spanning trees to check whether the spanning tree is distinct spanning tree or not. Also, since we can filter symmetric spanning trees by filtering only VBVs using Constraints 1,2,3, and 4, less spanning trees are generated in our method. As shown in Table 9.1, a much smaller number of VBVs, which satisfy the constraints, are generated. And, by this small number of VBVs, we only need to generate a small number of spanning trees instead of enumerating spanning trees to enumerate all possible distinct spanning trees. In Table 9.1, to obtain all possible number of VBVs, first we compute the number of valid vertical bit patterns on each row, then the numbers for rows are multiplied together. In an MxN grid, since we have $2^M - 1$ valid bit patterns for each row, and we can have vertical edges in N-1 rows, the possible number of VBVs on the grid is $(2^M - 1)^{N-1}$.

	# VBVs satisfying Constraints 1,2,3, and 4	**#All possible VBVs**
3x3	11	49
4x4	333	3,375
5x5	95,647	923,521
6x6	65,698,850	992,436,543

Table 9.1: Comparison between the number of VBVs satisfying Constraints 1,2,3, and 4 and the number of all possible VBVs in each size of grid.

Chapter 10: Analysis of Enumerated Space of Distinct Spanning Trees

In Chapter 9, our distinct spanning tree enumeration algorithm was introduced. Using the algorithm, we can obtain an enumerated space of distinct spanning trees. In this chapter, we analyze the enumerated space in 3x3, 4x4, 5x5, and 6x6 grids. In the next chapter, this analysis will be used as a standard, where analyses of sampling spaces of existing spanning tree generation algorithms are compared. The enumeration process ran in parallel using cloud computing with up to 500 CPUs. A 6x6 grid took three days with our enumeration despite the $O(2^{36})$ complexity.

10.1 Number of Distinct Spanning Trees

In this section, we show the number of all possible distinct spanning trees in each grid. In Table 1, the number of all possible spanning trees, including duplicated spanning trees, is also provided as an upper bound. To determine the number of all possible spanning trees in a grid graph, Kirchhoffs matrix tree theorem [16] was used. In Table 10.1, we can see that the number of all possible distinct spanning trees is almost $\frac{1}{8}$ of the number of all possible spanning trees in each grid. This is because we enumerate distinct spanning trees, in which each distinct one is one of its symmetric group which has at most eight spanning trees. These numbers we calculated is using our new enumeration algorithm.

	# Distinct Spanning Trees	# Spanning Trees
3x3	28	192
4x4	12,600	100,352
5x5	69,699,849	557,568,000
6x6	4,070,693,024,640	32,565,539,635,200

Table 10.1: The number of all possible distinct spanning trees and number of all possible spanning trees in a corresponding grid size.

10.2 Analysis of Metrics

In this section, we analyze the enumerated set of distinct spanning trees in terms of maze metrics. For our purposes, we use #Turns, #Straights, #T-Junctions, #Cross-Junctions, and #Terminals. In each metric, we investigate how the distinct spanning trees cover values of a metric.

10.2.1 Turns

Range of #Turns

In Table 10.2, we can see the maximum values and the minimum values of #Turns that spanning trees can have in 3x3, 4x4, 5x5, and 6x6 grids. As provided in Table 10.2, in each grid, spanning trees can have no turns but cannot have turns on all nodes. For example, in a 6x6 grid, the number of nodes is 36, but the maximum value of #Turns is 30, which is 83% of all nodes of a grid.

Histogram of #Turns

Table 10.2 shows a range of metric values but does not show which metric values are covered more frequently by the enumerated space. Now, we provide a histogram of #Turns, which shows the number of distinct spanning trees that have a corresponding metric value. Figure 10.1 shows a 1D histogram of #Turns. As shown in Figure 10.1, in each grid, #Turns

	Min	Max
3x3	0	5
4x4	0	12
5x5	0	19
6x6	0	30

Table 10.2: Minimum (Min) and maximum (Max) values of #Turns in each grid size.

has a bell curve shape of distribution. It indicates that spanning trees with middle values of #Turns are very easy to have, but spanning trees with end values of #Turns are very hard to have. For example, in a 6x6 grid, the number of spanning trees with full of turns (#Turns=30), such as a Hilbert curve [2], is close to 0, which means that it is very hard to find the one in the enumerated space. Also, we can see that when a grid size gets larger, a distribution slightly leans toward left; a metric value slightly less than a middle value has the highest frequency in histograms. For example, in a histogram of a 3x3 grid, where #Turns=2 and #Turns=3 are middle values, #Turns=3 has he highest frequency. But, in a histogram of a 6x6 grid, where #Turns=15 is a middle value, #Turns=10 has the highest frequency.

10.2.2 Straights

Range of #Straights

In Table 10.3, we can see a range of #Straights by looking at maximum values and the minimum values of #Straights that spanning trees can have in 3x3, 4x4, 5x5, and 6x6 grids. Like values of #Turns, in each grid, we can have spanning trees with no straights. But, we cannot have spanning trees where all nodes have straight cells. Table 10.3 shows that in a 6x6 grid, the maximum number of straights is 24, which is 66% of all cells on a grid. This maximum number is also less than the maximum number of #Turns.

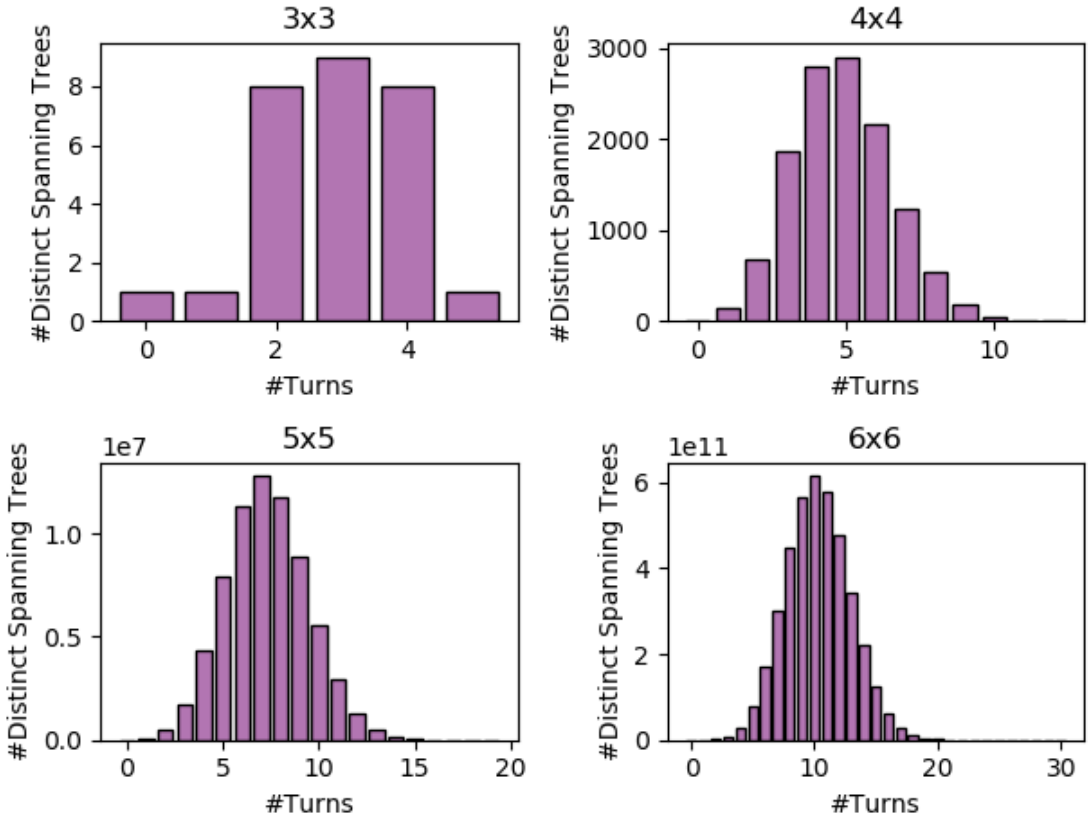


Figure 10.1: Histograms that show how many distinct spanning trees have corresponding values of #Turns in corresponding grid size. As shown in each histogram, X axis and Y axis denote the metric value and the number of distinct spanning trees respectively.

	Min	**Max**
3x3	0	3
4x4	0	6
5x5	0	15
6x6	0	24

Table 10.3: Minimum (Min) and maximum (Max) values of #Straights in each grid size.

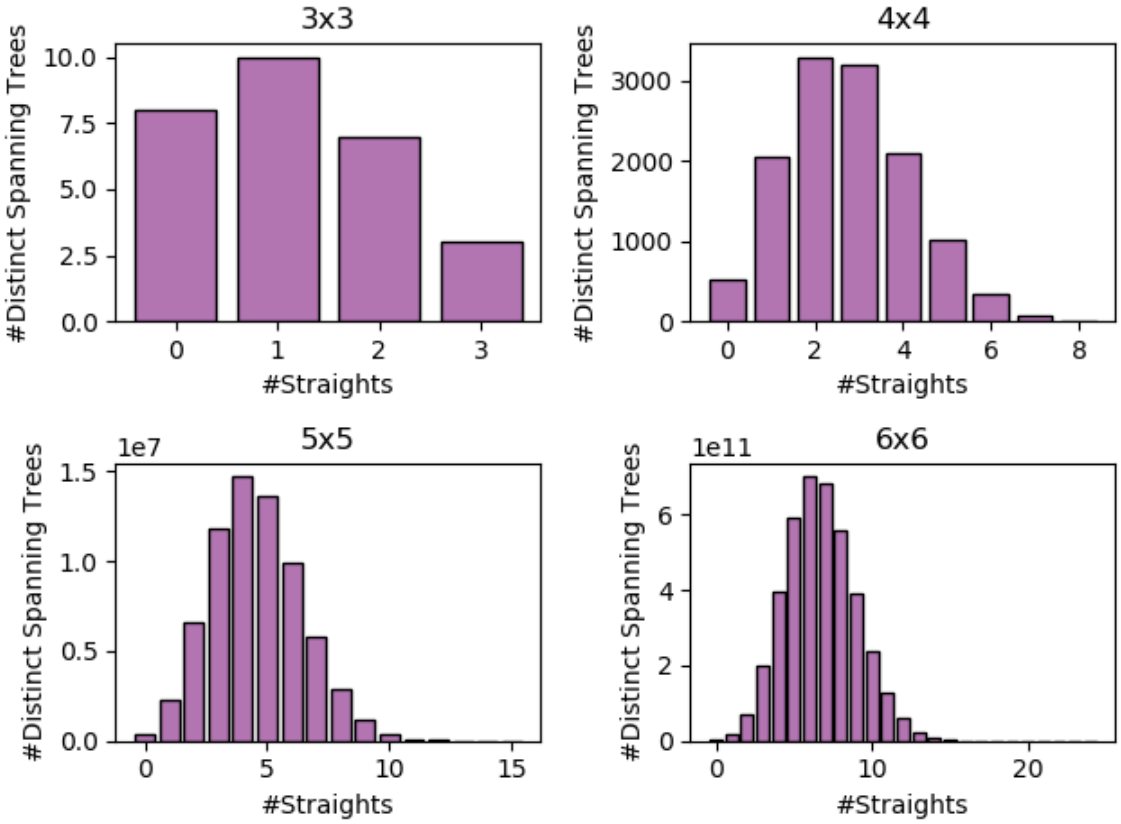


Figure 10.2: Histograms that show how many distinct spanning trees have corresponding values of #Straights in corresponding grid size.

Histogram of #Straights

Figure 10.2 shows a histogram of # Straights in each grid. Like #Turns, they have a bell curve shape of distribution. Spanning trees with middle values are easier to find than spanning trees with end values. In a 6x6 grid, we can see that the number of spanning trees with the maximum #Straights is close to 0, which means that it is very hard to find spanning trees with full of straights in the enumerated space. Additionally, Figure 10.2 shows that all histograms lean slightly toward left, which means that they have the highest frequencies on metric values slightly less than middle values.

	Min	Max
3x3	0	3
4x4	0	6
5x5	0	11
6x6	0	16

Table 10.4: Minimum (Min) and maximum (Max) values of #T-Junctions in each grid size.

10.2.3 T-Junctions

Range of #T-Junctions

In Table 10.4, a range of #T-Junctions in each grid is provided with the minimum value and the maximum value. It is possible to have spanning trees with no T-Junction. Also, we can see that it is not possible to have spanning trees where all cells of a grid are T-Junction cells. For example in a 6x6 grid, a spanning tree can have #T-Junctions = 16 at most, which is 44% of total number of cells on a grid. This number is a lot less than the maximum values of #Turns and #Straights.

Histogram of #T-Junctions

To represent frequencies of spanning trees based on #T-Junctions, we show histograms of #T-Junctions in Figure 10.3. Each histogram has a bell curve shape of distribution. From the histograms, we can see that in each grid, spanning trees with no T-junction or full of T-junctions are very hard to find compared to spanning trees with middle values of #T-Junctions. Unlike #Turns and #Straights, the histograms do not lean toward left. Middle values of #T-Junctions have the highest frequencies in the histograms.

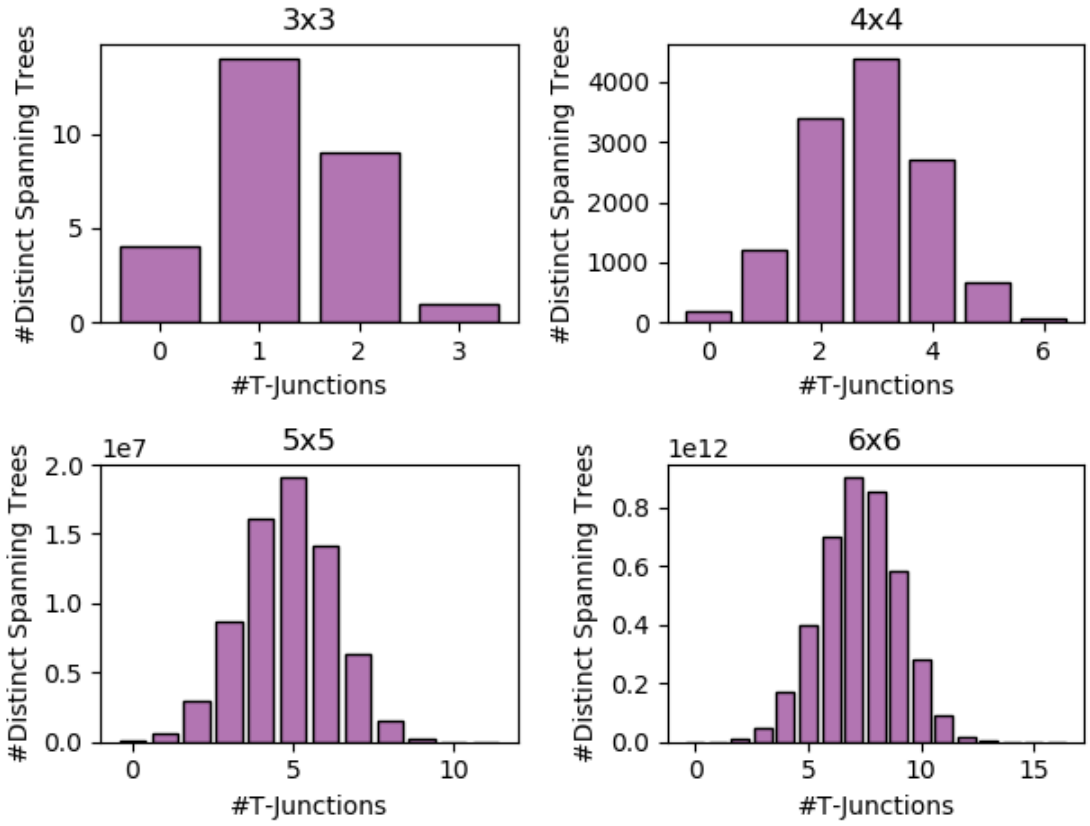


Figure 10.3: Histograms that show how many distinct spanning trees have corresponding values of #T-Junctions in corresponding grid size.

	Min	Max
3x3	0	1
4x4	0	2
5x5	0	4
6x6	0	8

Table 10.5: Minimum (Min) and maximum (Max) values of #Cross-Junctions in each grid size.

10.2.4 Cross-Junctions

Range of #Cross-Junctions

Table 10.5 represents a range of #Cross-Junctions by the minimum value and maximum value in each grid. We can have a spanning tree with no cross-junctions. Also, there is no case in which all cells of a grid are Cross-Junction cells. A shown in Table 10.5, in a 6x6 grid, at most 8 cells can be Cross-Junction cells at the same time. This number is 22% of all cells of a grid. The maximum value of #Cross-Junctions is a lot less than maximum values of #Turns, #Straights, and #T-Junctions. Since a cross-junction cell needs four paths from its center, the cell cannot be placed on boundaries of a grid unlike other cells. That is why #Cross-Junctions has maximum value less than other metrics.

Histogram of #Cross-Junctions

In Figure 10.4, we represent a histogram of #Cross-Junctions in each grid. As shown in Figure 10.4, while other metrics have bell curve shape of distributions, #Cross-Junctions has a skewed right shape of distribution. Most of spanning trees have 0 or 1 cross-junction. It is almost impossible to have two cross-junctions in a 4x4 grid, four cross-junctions in a 5x5 grid, and eight cross-junctions in a 6x6 grid.

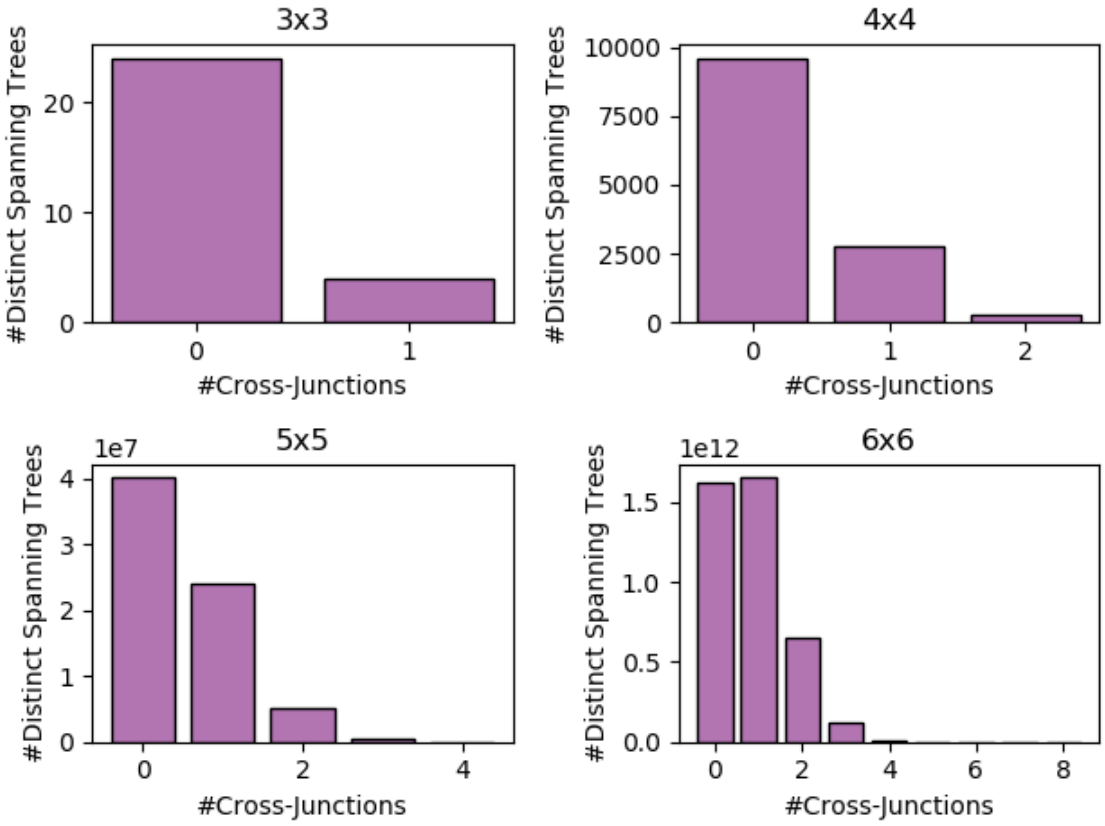


Figure 10.4: Histograms that show how many distinct spanning trees have corresponding values of #Cross-Junctions in corresponding grid size.

	Min	Max
3x3	2	6
4x4	2	9
5x5	2	14
6x6	2	22

Table 10.6: Minimum (Min) and maximum (Max) values of #Terminals in each grid size.

10.2.5 Terminals

Range of #Terminals

Table 10.6 show ranges of #Terminals. As shown in Table 10.6, there is no spanning tree, which has no terminal on it. If there is no terminal on a grid, it denotes that the grid has a loop. At least two terminals exist on spanning trees. The spanning trees with two terminals do not have junctions and are filled up with turns and straights. Like other metrics, we cannot have all cells of of a grid as Terminal cells. If all cells are Terminal cells, the resulting graph will be disconnect graph. We can see that in a 6x6 grid, the maximum value of #Terminals, which is 22, is 61% of all cells of a grid.

Histogram of #Terminals

Figure 10.5 shows histograms of #Terminals. These histograms have distributions with bell curve shape. Middle values of #terminals are easy to find compared to end values of #Terminals. We can call a maze with two terminals a labyrinth. By the histograms of #Terminals, we can see that labyrinths are very hard to find in the enumerated space. Also, a maze with full of terminals can be considered as a caterpillar graph [1]. In the histograms, we can see that the caterpillar graphs are very hard to find. Like #T-junctions, distributions of histograms do not lean to left or right. The middle values of #Terminals have the highest frequencies in the histograms.

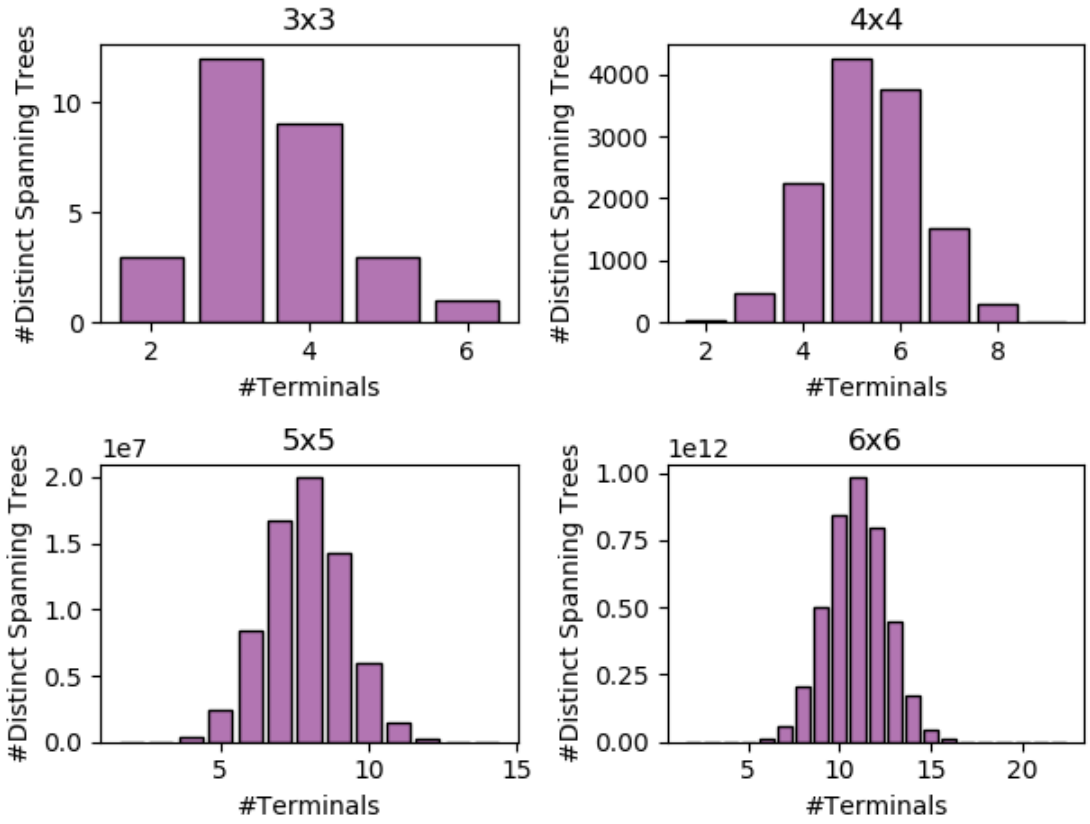


Figure 10.5: Histograms that show how many distinct spanning trees have corresponding values of #Terminals in corresponding grid size.

10.3 Analysis of Pairwise Metrics

In the previous section, we analyzed spanning trees with single metric. In this section, we analyze spanning trees with pairwise metrics, such as #Turns & #Straights or #T-Junctions & #Terminals so that we can see how many spanning trees have each 2D combination of metrics. Also, we hope to see some signatures of relationship between metrics.

10.3.1 Histograms of Pairwise Metrics

In Figures 10.6 and 10.7, we also provide 2d histograms of distinct spanning trees, where the number of distinct spanning trees per 2D combination of metric values are denoted. In the 2D histograms, darker color denotes more spanning trees in the corresponding spot.

In Figures 10.6 and 10.7, when one of the 2D metrics is #Cross-Junctions, a shape of distribution is skewed toward the low value of #Cross-Junctions. In other cases, a shape of distribution tends to be in the middle of a chart space, which indicates that there are many spanning trees that have somewhat middle values of the associated metrics.

In Figure 10.7(c), histograms of #Terminals and #T-Junctions have bumpy textures. When there is one T-junction, #Terminals is raised by 1, and, when there is one cross-junction, #Terminals is raised by 2. Thus, when #Terminals is an odd value, and #T-Junctions is an even value, there is no spanning tree with the corresponding metric values. Also, when #Terminals is an even value, and #T-Junctions is odd value, there is no spanning tree with the associated two metrics values. We can write the equation regarding these three metrics like the below.

$$\#TJunctions + 2 * \#CrossJunctions = \#Terminals - 2 \tag{10.1}$$

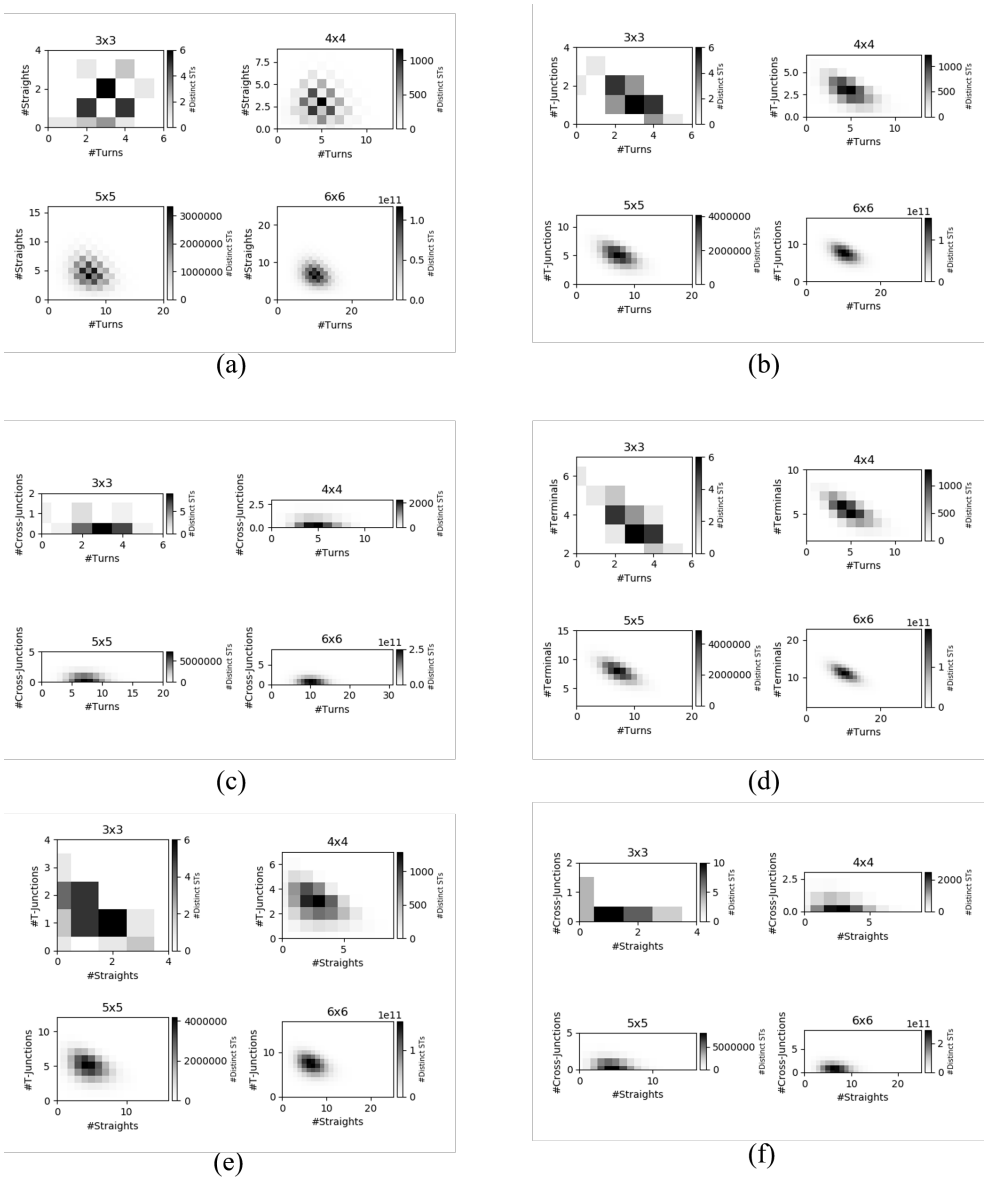


Figure 10.6: 2D histograms that show how many distinct spanning trees have corresponding 2D combination of metric values ((a)#Turns & #Straights, (b)#Turns & #T-Junctions, (c)#Turns & #Cross-Junctions, (d)#Turns & #Terminals, (e)#Straights & #T-Junctions, (f)#Straights & #Cross-Junctions) in corresponding grid size. The number of distinct spanning trees is represented by different brightness. Darker color indicates more spanning trees.

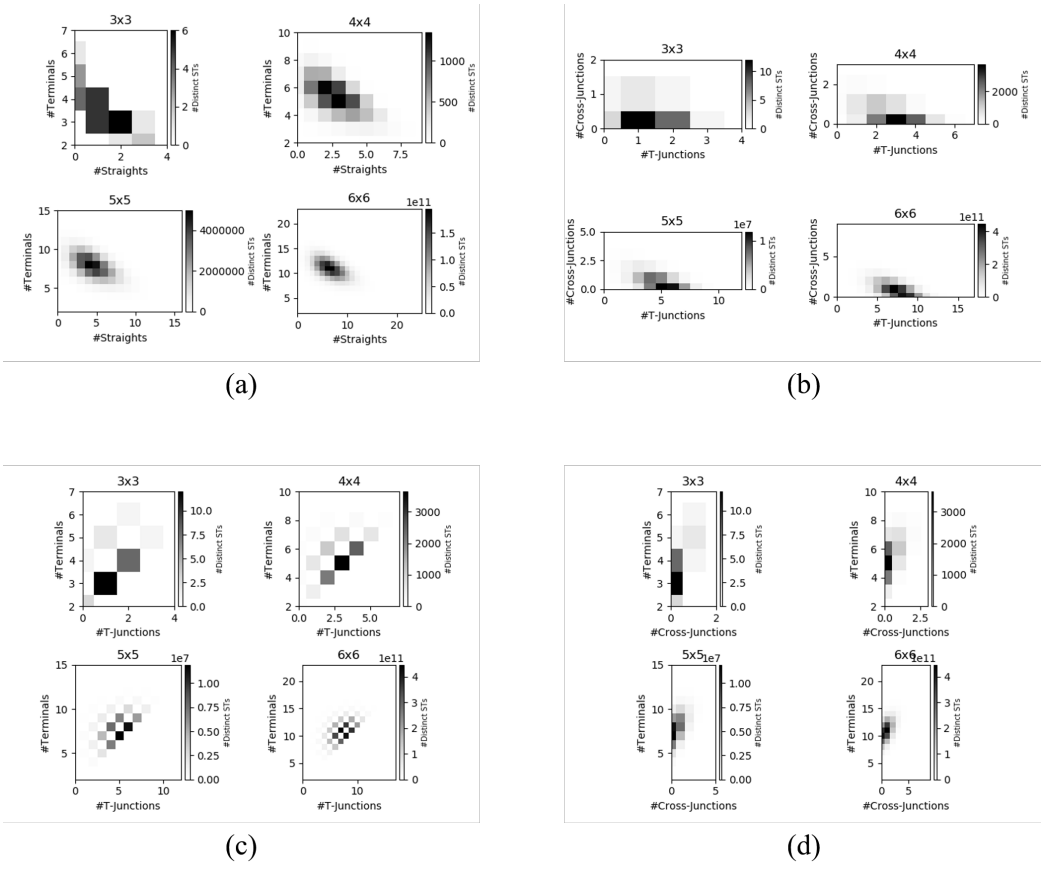


Figure 10.7: 2D histograms that show how many distinct spanning trees have corresponding 2D combination of metric values ((a)#Straights & #Terminals, (b)#T-Junctions & #Cross-Junctions, (c)#T-Junctions & #Terminals, (d)#Cross-Junctions & #Terminals) in corresponding grid size.

Since a spanning tree has two terminals when there are no junctions, right side of Equation 10.1 is #Terminals 2 so that we have 2 for Number of Terminals in the equation when #T-Junctions = 0 and #Cross-Junctions = 0. When #Terminals is an even value, and #T-Junctions is an odd value, by Equation 10.1, and 2 × #Cross-Junctions cannot be odd number. Hence, spanning trees with an even number of terminals and odd number of T-junctions cannot exist. Likewise, spanning tree with odd number of terminals and even number of T-junctions cannot exist. It lets us know the relationship among these metrics, #T-Junctions, #Cross-Junctions, and #Terminals.

10.4 Analysis of Metric Vectors

In previous sections, we analyzed spanning trees with single metric and pairwise metrics. In this section, we analyze spanning trees in terms of metric vector, which is a vector of spanning tree metrics. Note that each metric vector is represented as a 5-tuple of (#Turns, #Straights, #T-Junctions, #Cross-Junctions, #Terminals). Since a metric vector is 5D data, it is hard to visualize the data compared to single metric and pairwise metric. Thus, we do not show histograms in this section.

10.4.1 Number of Metric Vectors

Although each spanning tree has one metric vector, spanning trees can have the same metric vector. Thus, the number of metric vectors can be less than the number of spanning trees. Table 10.7 shows the number of metric vectors in each grid. In Table 10.7, we can see that the number of metric vectors is much less than the number of distinct spanning trees in Table 10.1. For example, a 6x6 grid has over 4 trillion distinct spanning trees, yet there are only 1,273 metric vectors. In a 5D space of metrics of a 6x6 grid, the possible number of metric vectors is 41,515. This number is computed by calculating the possible

	#Metric Vectors
3x3	11
4x4	90
5x5	385
6x6	1,273

Table 10.7: The number of unique metrics vectors in a grid size of 3x3 up to 6x6.

number of combinations of metric values, where #Turns + #Straights + #T-Junctions + #Cross-Junctions + #Terminals = 36. Then, we can see that the actual number of metric vectors, 1,273, is still very small compared to the possible number of metrics vectors, 41,515.

From these numbers, we can see that when we visualize spanning trees over the metric vector space, spanning trees with the same metric vector will be represented as a single point, and we will have a few points over the space. As shown in Figure 10.8, we can have three type of distribution with a small number of points. In 1D histograms of metrics shown in Section 10.2, there is no missing values between the minimum value and maximum value of each metrics. Thus, we can conclude that spanning trees over the metric vector space are closed together continuously as shown in Figure 10.8.

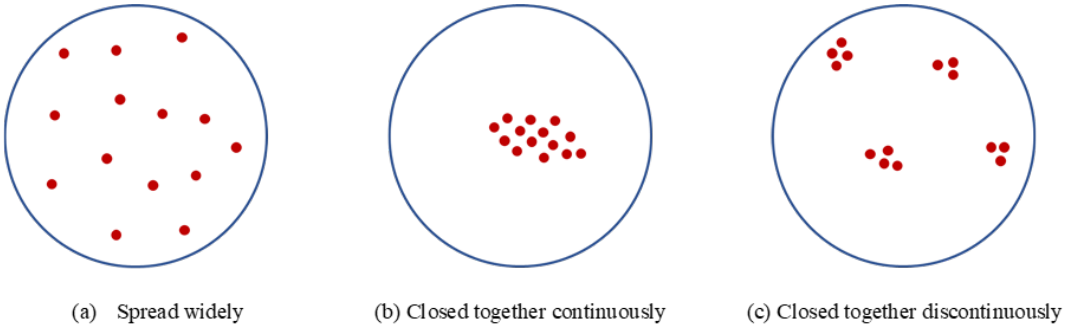


Figure 10.8: Figures that showing several distribution types.

Chapter 11: Analysis of Sampling Space of Existing Spanning Tree Algorithms

In this chapter, we analyze spanning trees sampled by spanning tree generation algorithms. Among the algorithms, we especially analyze the recursive backtracker, Prims algorithm, and Kruskals algorithm. In analyses of the algorithms, we sample 10,000,000,000 spanning trees randomly in 6x6 grid for each sampling algorithm. Please note that our research allowed duplicates in this sampling process, since it is hard to avoid these duplicates with these algorithms. As we did in Chapter 10, first, we show how many unique metric vectors each sampling space has.

11.1 Number of Metric Vectors

In this section, we determine the number of unique metric vectors sampled by each sampling algorithm. In Table 11.1, we provide the number of unique metric vectors in each sampling space.

	#Metric Vectors
Recursive Backtracker	303
Prim's	1107
Kruskal's	1124

Table 11.1: Number of metric vectors of each algorithm in a 6x6 grid with 10 billion samples.

In Table 11.1, we can see that, even with an enormous number of samples, these algorithms have much fewer metric vectors than the actual number of metric vectors on a 6x6 grid, which is 1,273. Especially, the recursive backtracker covers only 24% of the actual metric vector space. It means the recursive backtracker is limited in the mazes it can produces. In the next section, we investigate the range of metrics of each generation algorithm to see the distribution of sampled metric vectors. Also, for each algorithm, we see that which metric values are not sampled.

11.2 Range of Metrics

In this section, we provide a range of each metric in each algorithm in Table 11.2. To show a range in each algorithm, we provide maximum value and minimum value of the range. We also check if there are missing metric values. We compare the range of metrics between random sampling spaces and the enumerated space. This comparison shows which metric values in the enumerated space are sampled by the generation algorithms.

For all metrics, the metric values are contiguous. Next, we look at a range of each metric of each algorithm in detail so that we know which metric values in the enumerated space are sampled by each algorithm.

In Table 11.2(a), the recursive backtracker does not have #Turns less than 4. Prims algorithm and Kruskal's algorithm do not have higher values of #Turns compared to the recursive backtracker. Prims algorithm has #Turns not more than 24, and Kruskal's algorithm has #Turns not more than 28.

In Table 11.2(b), the recursive backtracker has all possible values of #Straights. However, other algorithms do not have high values of #Straights. Prims algorithm does not

	Min	Max	Missing Metric Values
Recursive Backtracker	4	30	∅
Prim's	0	24	∅
Kruskal's	0	28	∅
Enumeration	0	30	∅

(a) Number of Turns

	Min	Max	Missing Metric Values
Recursive Backtracker	0	24	∅
Prim's	0	23	∅
Kruskal's	0	22	∅
Enumeration	0	34	∅

(b) Number of Straights

	Min	Max	Missing Metric Values
Recursive Backtracker	0	9	∅
Prim's	0	16	∅
Kruskal's	0	16	∅
Enumeration	0	16	∅

(c) Number of T-Junctions

	Min	Max	Missing Metric Values
Recursive Backtracker	0	2	∅
Prim's	0	8	∅
Kruskal's	0	8	∅
Enumeration	0	8	∅

(d) Number of Cross-Junctions

	Min	Max	Missing Metric Values
Recursive Backtracker	2	11	∅
Prim's	4	22	∅
Kruskal's	2	21	∅
Enumeration	2	22	∅

(e) Number of Terminals

Table 11.2: Minimum (Min), maximum (Max) values, and missing metric values of each metric of each sampling algorithm. It also shows the values of the enumerated space (Enumeration). Missing metric values show values of corresponding metric that spanning trees do not have between corresponding Min and Max. ∅ in the column of missing metric values denotes that there is no missing metric value between corresponding Min and Max in a 6x6 grid.

have #Straights more than 23, and Kruskal's algorithm does not have #Straights more than 22.

In Table 11.2(c), the recursive backtracker does not have high values of #T-Junctions. It does not have #T-Junctions more than 9. Other algorithms have all possible values of #T-Junctions.

In Table 11.2(d), the recursive backtracker has only small values of #Cross-Junctions, and other algorithms have all possible values of #Cross-Junctions. The recursive backtracker has #Cross-Junctions less than 3.

In Table 11.2(e), the recursive backtracker does not have #Terminals more than half of the range of the metric of the enumerated space. It has #Terminals less than 12. Prim's algorithm does not have low values of #Terminals, which are #Terminals = 2 and #Terminals = 3. Kruskal's algorithm does not have only the maximum value of #Terminals of the enumerated space, which is #Terminals = 22.

From the above analyses, we can see that it is hard to generate spanning trees with lots of branches using the recursive backtracker, which have higher values of #T-Junctions or #Cross-Junctions, and result in a higher of #Terminals. In Prim's algorithm and Kruskal's algorithm, they usually have wider ranges of metrics than the recursive backtrackers. But, generating spanning trees with straight passages and lots of turns is more difficult.

In the next section, to visualize which metric values are sampled more frequently than others in each algorithm, we show histograms of metrics in the next section.

11.3 Histograms of Metrics

In Figure 11.1, each figure shows histograms of each metric of all generation algorithms using bar charts. Each histogram shows how many spanning trees of each algorithm have

the corresponding metric value in a 6x6 grid. Histograms with red bars, yellow bars, and blue bars are the histograms of the recursive backtracker, Prims algorithm, and Kruskals algorithm, respectively. Since histograms of the algorithms are placed in the same figure, we can also compare sampling spaces of the algorithms easily.

As we can see in Figure 11.1, algorithms have different signatures. The recursive backtracker algorithm generally has high values in metrics #Turns, and #Straights and low values in metrics #T-Junctions, #cross-Junctions, and #Terminals. This means that the recursive backtracker tends to generate spanning trees with long windy passages and less branches. Prims and Kruskals algorithms usually have high values in #T-Junctions, #Cross-Junctions, and #Terminals and low values in #Turns and #Straights. This means that both algorithms tend to generate spanning trees with many branches. When we compare Prims and Kruskal's algorithms, Prims algorithm is shifted compared to Kruskal's algorithm. For example, in Figure 11.1(a), both Prims and Kruskals algorithms have small values in #Turns, but Prims algorithm tends to have smaller values of #Turns than Kruskals algorithm.

To see if there are other signatures among the algorithms, we show histograms of two metrics in Figures 11.2, 11.3, and 11.4. In these histograms, we can determine how many spanning trees have corresponding pair-wise metric values. Since there are $\binom{5}{2} = 10$ pairs of two metrics, there are a total of ten histograms of two metrics in Figures 11.2, 11.3, and 11.4. In each histogram, the amount of spanning trees is distinguished by darkness. A spot with a darker color in a 2D histogram denotes that there are more spanning trees with the associated metric values. Likewise, when a plot has brighter color in a 2D histogram, it indicates that less spanning trees exist in that spot.

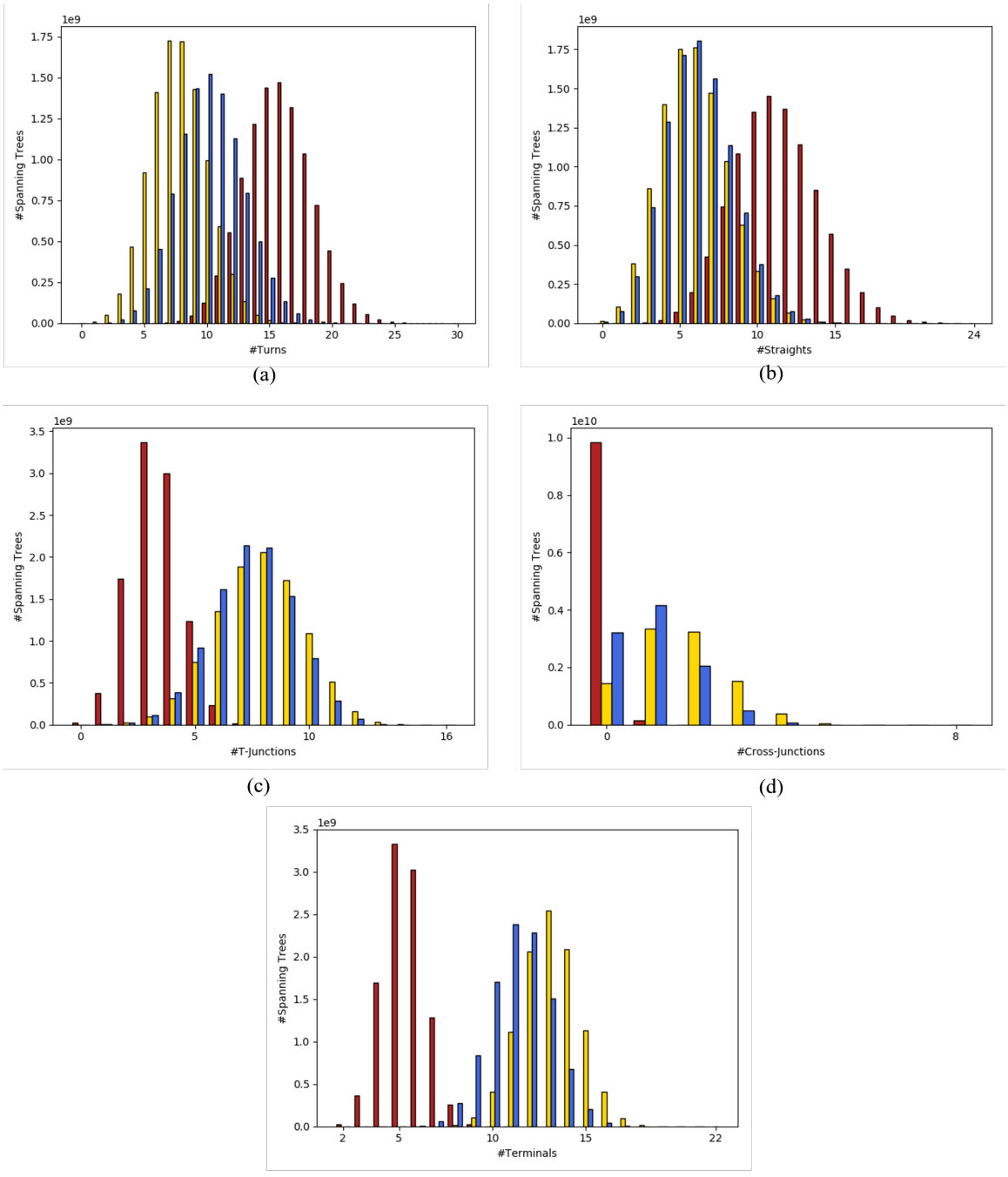


Figure 11.1: Histograms that show how many spanning trees sampled by each generation algorithm (red: recursive backtracker, yellow: Prim's, blue: Kruskal's) have corresponding metric values ((a)#Turns, (b)#Straights, (c)#T-Junctions, (d)#Cross-Junctions, (e)#Terminals) in a 6x6 grid. As shown in each figure, X axis and Y axis denote the metric value and the number of sampled spanning trees having that metric values respectively.

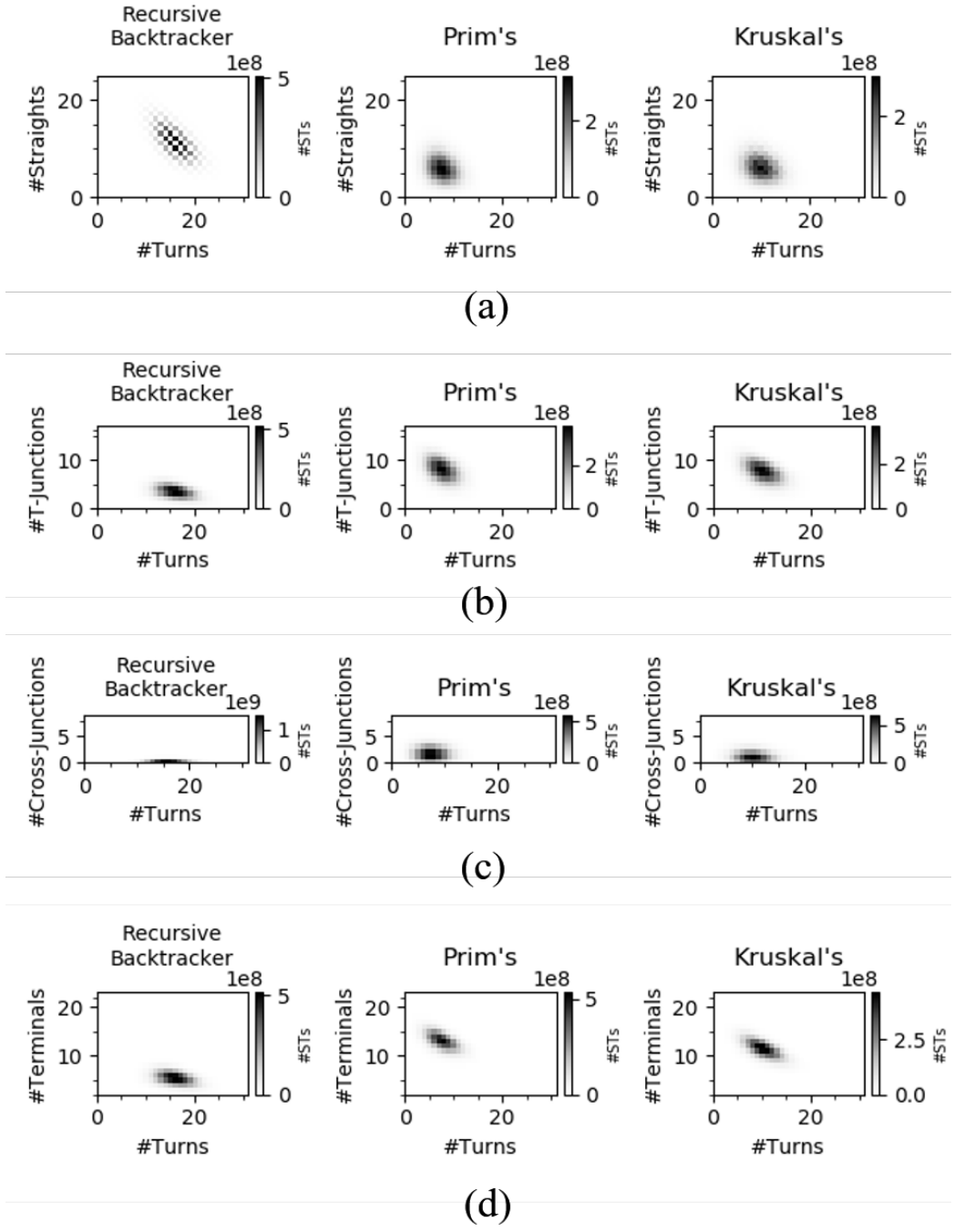


Figure 11.2: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#Turns & #Straights, (b)#Turns & #T-Junctions, (c)#Turns & #Cross-Junctions, (d)#Turns & #Terminals) by corresponding algorithms in a 6x6 grid. The number of distinct spanning trees is represented by different brightness. Darker color indicates more spanning trees.

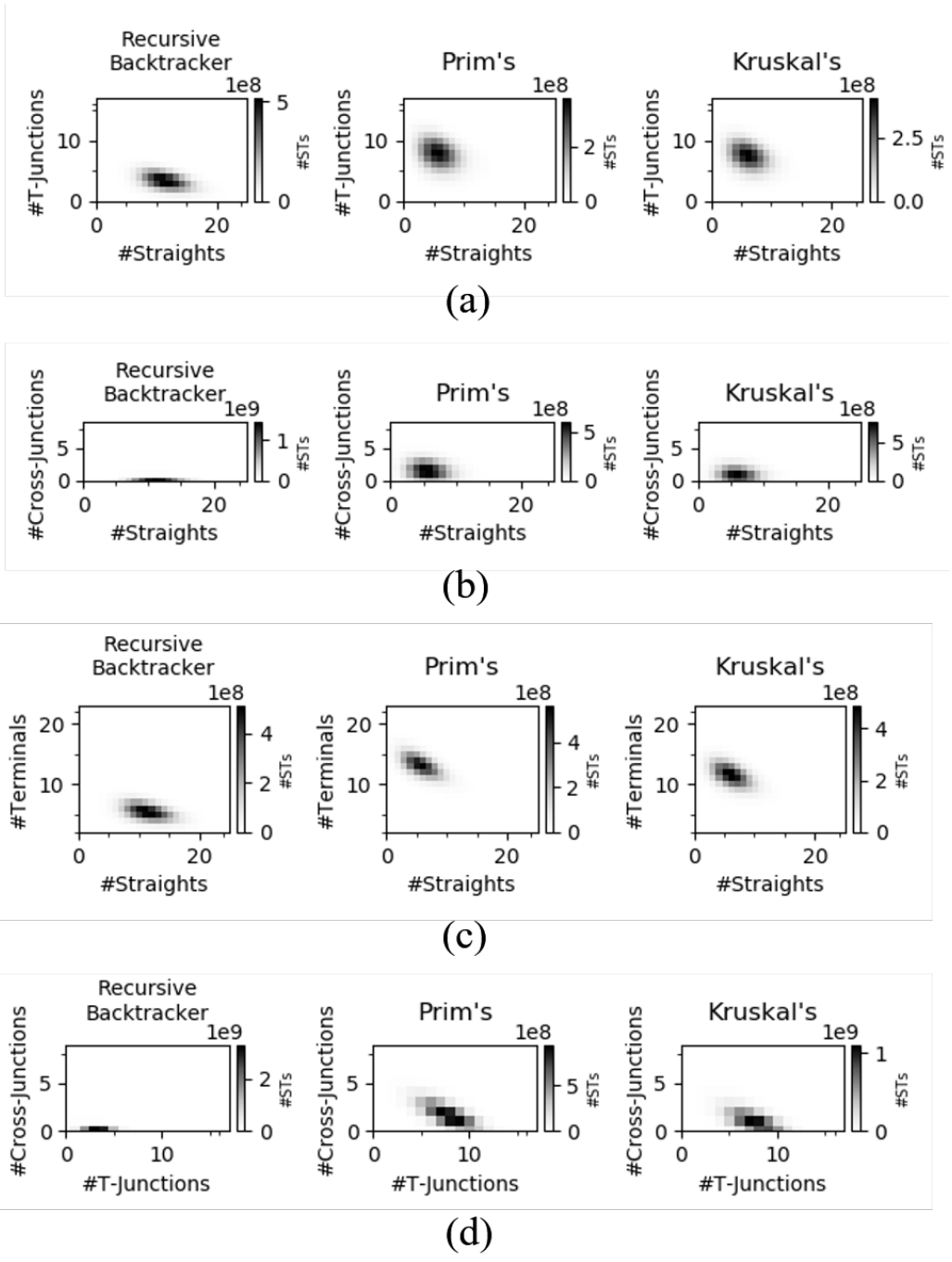


Figure 11.3: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#Straights & #T-Junctions, (b)#Straights & #Cross-Junctions, (c)#Straights & #Terminals, (d)#T-Junctions & #Cross-Junctions) by corresponding algorithms in a 6x6 grid.

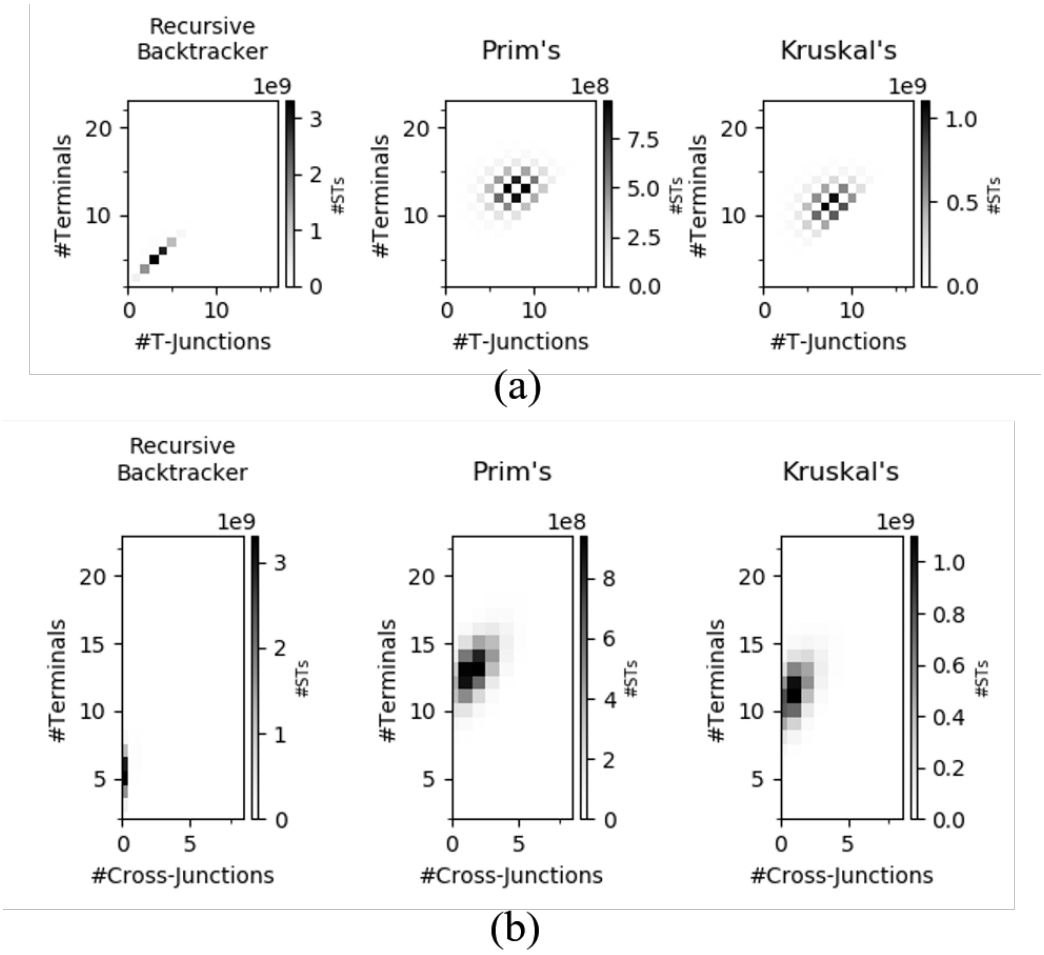


Figure 11.4: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#T-Junctions & #Terminals, (b)#Cross-Junctions & #Terminals) by corresponding algorithms in a 6x6 grid.

In Figures 11.2, 11.3, and 11.4, we can see that the recursive backtracker usually has the smaller area of sampled spanning trees than the areas of other algorithms. Especially, when one of metrics in a histogram is #Cross-Junctions, since few cross-junctions are sampled by the recursive backtracker, the algorithm has much smaller area of sampled spanning trees than other algorithms. It denotes that, in 6x6 grid, Prims and Kruskals algorithm have higher probability to sample spanning trees with wider range of metric values than the recursive backtracker. Also, in the histograms, we can see that Prims and Kruskals algorithms have similar shape of area of sampled spanning trees compared to the recursive backtracker. It may represent that both algorithms have similar features on their sampled spanning trees.

In Figure 11.2(a), which are histograms of #Turns and #Straights, we can see the bumpy texture only on the map of the recursive backtracker. This histogram is provided numerically in Figure 11.5 so that we can see the exact number of spanning trees that have the corresponding two metrics. As shown in Figure 11.5, the bumpy texture is shown on the map because, when both #Turns and #Straights have even values or odd values, the histogram has larger number of spanning trees. And, when one metric has even value and the other metric has odd value, the histogram has smaller number of spanning trees. Since a total value of all five metrics should be 36 in a 6x6 grid, when a total value of the two metrics, #Turns and #Straights, is an odd value, a total value of the rest metrics, #T-Junctions, #Cross-Junctions, and #Terminals, should be odd. It indicates that a spanning tree should have a cross-junction. Since a cross-junction creates two more terminals on a spanning tree, a total value of #T-Junctions, #Cross-Junctions, and #Terminals is raised by 3 per cross-junction (1 cross-junction + 2 terminals). However, when we have

Number of Turns	Number of Straights																								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1178	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	916	0	2609	0	4854	0	2965	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7796	0	39944	0	80319	0	67738	0	23396
7	0	0	0	0	0	0	0	0	0	0	0	0	0	13218	0	138309	0	513255	0	749448	0	542921	0	156113	0
8	0	0	0	0	0	0	0	0	0	0	6053	0	138776	1426	951391	6069	2785664	7747	3952885	4736	2750605	0	820055	0	71990
9	0	0	0	0	0	0	0	0	0	28771	683	566935	28874	3612558	86088	10484926	91357	14801991	34704	10331533	5944	3171497	0	299902	0
10	0	0	0	0	0	0	0	75007	8232	1516046	136981	10089419	383668	29586050	440665	42448687	197175	30092965	35064	9296306	1053	956033	0	33655	
11	0	0	0	0	0	679	0	120582	21352	2880522	348376	21383083	1103651	66597980	1423952	99116743	758131	70584265	154035	22099536	4309	2562065	0	84906	0
12	0	0	0	0	499	0	119390	37685	4034695	642733	35159357	2444456	121571865	3331971	191736505	1945646	139949018	416027	45240801	22157	5737210	0	206794	0	0
13	0	0	0	0	0	79212	31869	3971947	812843	45251454	3863215	181317471	6075029	310914135	3998557	238023350	865588	80396098	39217	10920800	0	454230	0	0	0
14	0	0	0	0	27824	13256	2889664	772670	45608195	4879139	220878048	9021328	426938976	6443192	353026373	1548919	125057427	81290	17984110	0	770266	0	0	0	0
15	0	0	0	7412	2762	1469229	542981	35189929	4744886	219118951	10645533	496057469	8564831	455482196	2315410	173646168	114527	26547019	0	1242251	0	0	0	0	0
16	0	0	944	692	546089	247199	20960247	3449115	176313153	10257734	484679849	9546832	510169634	2758462	216594448	186922	34977208	0	1712401	0	0	0	0	0	0
17	0	0	0	161244	71233	9378170	1907698	117728478	7901968	394200854	8812036	493852323	2846065	240633073	210191	43058162	0	2260518	0	0	0	0	0	0	0
18	0	0	29626	14187	3108668	759108	56908213	4642483	265557941	6814357	410179532	2573805	234834243	214351	48242844	0	2751725	0	0	0	0	0	0	0	0
19	0	4495	1644	770081	181664	22304575	2120145	145538753	4448539	289662024	1980420	201553549	174583	48420783	0	3097053	0	0	0	0	0	0	0	0	0
20	0	0	136128	25262	6549369	721036	64325623	7254208	172709235	1315254	149773298	111006	42634797	0	3157965	0	0	0	0	0	0	0	0	0	0
21	0	13225	826	1453357	154708	22426339	902493	85852165	785798	95865038	81964	32945863	0	2839877	0	0	0	0	0	0	0	0	0	0	0
22	336	0	235562	25046	5864771	270733	34938325	371558	52941663	40536	22363244	0	2250309	0	0	0	0	0	0	0	0	0	0	0	0
23	0	26418	741	1163811	38433	11253580	132539	24849275	22681	13286575	0	1608664	0	0	0	0	0	0	0	0	0	0	0	0	0
24	2183	238	147724	3605	2704531	40008	9635434	10173	6975954	0	1001434	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	16318	0	460441	6098	2949208	2123	3168062	0	527767	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	1021	0	50340	725	640308	592	1281221	0	347269	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	3547	0	95701	0	393325	0	153569	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	9615	0	80348	0	50295	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	7382	0	23922	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	5735	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 11.5: Table that showing the histogram of the recursive backtracker in Figure 11.2(a) numerically.

a T-junction, since it creates one more terminal on a spanning tree, a total value of #T-Junctions, #Cross-Junctions, and #Terminals is raised by 2 (1 T-junction + 1 terminals), It indicates that we cannot have an odd value of three metrics #T-Junctions, #Cross-Junctions, and #Terminals without a cross-junction. Then, since spanning trees of the recursive backtracker unlikely have a cross-junction, the number of spanning trees, in which a total value of #Turns and #Straights is an odd value, is small. From this analysis, we can see that the recursive backtracker has low probability to have cross junctions on its sampled spanning trees.

In Figure 11.4(a), which are histograms of #Terminals and #T-Junctions, maps of all three algorithms have bumpy textures. It also happened in Figure 10.7(c), which are histograms of #Terminals and #T-Junctions in the enumerated space. We provide these histograms numerically in Figure 11.6. As shown in Figure 11.6, when #Terminals is an odd value, and #T-Junctions is an even value, there is no sampled spanning tree with the corresponding metric values. Also, when #Terminals is an even value, and #T-Junctions is

odd value, there is no sampled spanning tree with the associated two metrics values. Here, we provide the equation regarding these three metrics again.

$$\#TJunctions + 2 * \#CrossJunctions = \#Terminals - 2 \tag{11.1}$$

As explained in Section 10.3.1, when #Terminals is an even value, and #T-Junctions is an odd value, by Equation 11.1, and 2 $\times$ #Cross-Junctions cannot be odd number. Thus, we cannot have spanning trees with an even number of terminals and odd number of T-junctions. Likewise, spanning tree with odd number of terminals and even number of T-junctions cannot exist. The non-existent spanning trees yield bumpy textures in the histograms. Although this feature does not show unique signatures among spanning tree generation algorithms, it shows the relationship among these metrics, #T-Junctions, #Cross-Junctions, and #Terminals.

Having showed the signatures of each algorithm on a 6x6 grid, in the next section, we examine larger grid sizes, such as 20x20 grid to see how the characteristics of the algorithms are changed in the large grid. We provide histograms of one metric and histograms of two metrics in a 20x20 grid and investigate the histograms.

11.4 Histograms of Metrics in 20x20 grid

In this section, we sampled 10,000,000,000 spanning trees randomly for each algorithm in a 20x20 grid. From those sampled spanning trees, we obtained histograms of one metric as shown in Figure 11.7 and histograms of two metrics as shown in Figures 11.8 and 11.9.

As shown in Figure 11.7, in 20x20 grid, the recursive backtracker still tends to have high values in metrics, #Turns and #Straights. Prims and Kruskals algorithms also still tend to have high values in metrics, #T-Junctions, #Cross-Junctions, and #Terminals. In

		Number of Dead-Ends																				
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Number of T-Junctions	0	24580605	0	1317675	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	362914897	0	18151719	0	29569	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	2	0	0	1690282250	0	55038208	0	63444	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	3	0	0	0	3312688606	0	56861731	0	18200	0	0	0	0	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	2970696625	0	23834845	0	888	0	0	0	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	0	1229354627	0	3620547	0	0	0	0	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	231741062	0	116531	0	0	0	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	18221598	0	0	0	0	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	465195	0	0	0	0	0	0	0	0	0	0	0	0
	9	0	0	0	0	0	0	0	0	0	1178	0	0	0	0	0	0	0	0	0	0	0
	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Recursive Backtracker

		Number of Dead-Ends																				
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Number of T-Junctions	0	0	0	0	0	143	0	4217	0	39836	0	74682	0	33038	0	4275	0	308	0	0	0	0
	1	0	0	0	59	0	11537	0	218616	0	1043839	0	1101081	0	285403	0	29094	0	1062	0	0	0
	2	0	0	9	0	7828	0	393037	0	3980233	0	9634677	0	5568605	0	876123	0	57336	0	1394	0	0
	3	0	0	0	1484	0	261261	0	6182529	0	33413824	0	41804152	0	13198370	0	1256294	0	52161	0	647	0
	4	0	0	0	0	70309	0	4208798	0	51059517	0	144883030	0	94813873	0	16242106	0	910560	0	19371	0	60
	5	0	0	0	0	0	1286973	0	35882043	0	234607771	0	344264661	0	117354370	0	10818087	0	316888	0	2672	0
	6	0	0	0	0	0	0	11501184	0	175255829	0	617222127	0	461235049	0	80617003	0	3814798	0	50118	0	37
	7	0	0	0	0	0	0	0	58297116	0	508473260	0	941394914	0	351107347	0	30428154	0	672930	0	2209	0
	8	0	0	0	0	0	0	0	0	178312442	0	887059369	0	833271131	0	150107369	0	6081237	0	50907	0	29
	9	0	0	0	0	0	0	0	0	0	338333757	0	927375339	0	422274423	0	34935961	0	595074	0	1107	0
	10	0	0	0	0	0	0	0	0	0	0	399723403	0	571076288	0	118932137	0	4179537	0	23292	0	0
	11	0	0	0	0	0	0	0	0	0	0	0	289743094	0	200669246	0	17684750	0	229846	0	247	0
	12	0	0	0	0	0	0	0	0	0	0	0	0	124929922	0	38225325	0	1276209	0	4145	0	0
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	30541810	0	3657851	0	35642	0	0	0
	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3927480	0	150354	0	274	0	0
	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	233462	0	1869	0	0	0
	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5354	0	0	0	0

(b) Prim's

		Number of Dead-Ends																				
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Number of T-Junctions	0	5	0	562	0	12555	0	51941	0	75523	0	31745	0	4880	0	269	0	9	0	0	0	0
	1	0	840	0	43403	0	447729	0	1207005	0	1021501	0	276064	0	26265	0	944	0	20	0	0	0
	2	0	0	37978	0	1027156	0	6311735	0	10386377	0	5274501	0	860194	0	50209	0	1224	0	14	0	0
	3	0	0	0	723633	0	11995995	0	44435796	0	44357837	0	13357608	0	1305509	0	46896	0	773	0	0	0
	4	0	0	0	0	7710650	0	78382993	0	174939978	0	104221719	0	18379256	0	1051631	0	21892	0	167	0	0
	5	0	0	0	0	0	48972227	0	304699024	0	407040014	0	141691142	0	14323344	0	468281	0	5333	0	5	0
	6	0	0	0	0	0	0	193216372	0	731293788	0	574239663	0	113727857	0	6405377	0	112439	0	491	0	0
	7	0	0	0	0	0	0	0	485938002	0	1101911875	0	494949729	0	53918785	0	1621040	0	13519	0	9	0
	8	0	0	0	0	0	0	0	0	789630791	0	1045119682	0	258961709	0	14847599	0	220679	0	607	0	0
	9	0	0	0	0	0	0	0	0	0	829582700	0	618238417	0	80663728	0	2284456	0	14484	0	15	0
	10	0	0	0	0	0	0	0	0	0	0	558436854	0	223314283	0	14442598	0	183780	0	402	0	0
	11	0	0	0	0	0	0	0	0	0	0	0	235960759	0	47621110	0	1403926	0	6468	0	0	0
	12	0	0	0	0	0	0	0	0	0	0	0	0	60580459	0	5681842	0	66009	0	65	0	0
	13	0	0	0	0	0	0	0	0	0	0	0	0	0	9003195	0	348221	0	1193	0	0	0
	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	716392	0	9088	0	0	0	0
	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	26693	0	91	0	0	0
	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	408	0	0	0	0

(c) Kruskal's

Figure 11.6: Tables that showing the histograms of all sampling algorithms in Figure 11.3(c) numerically.

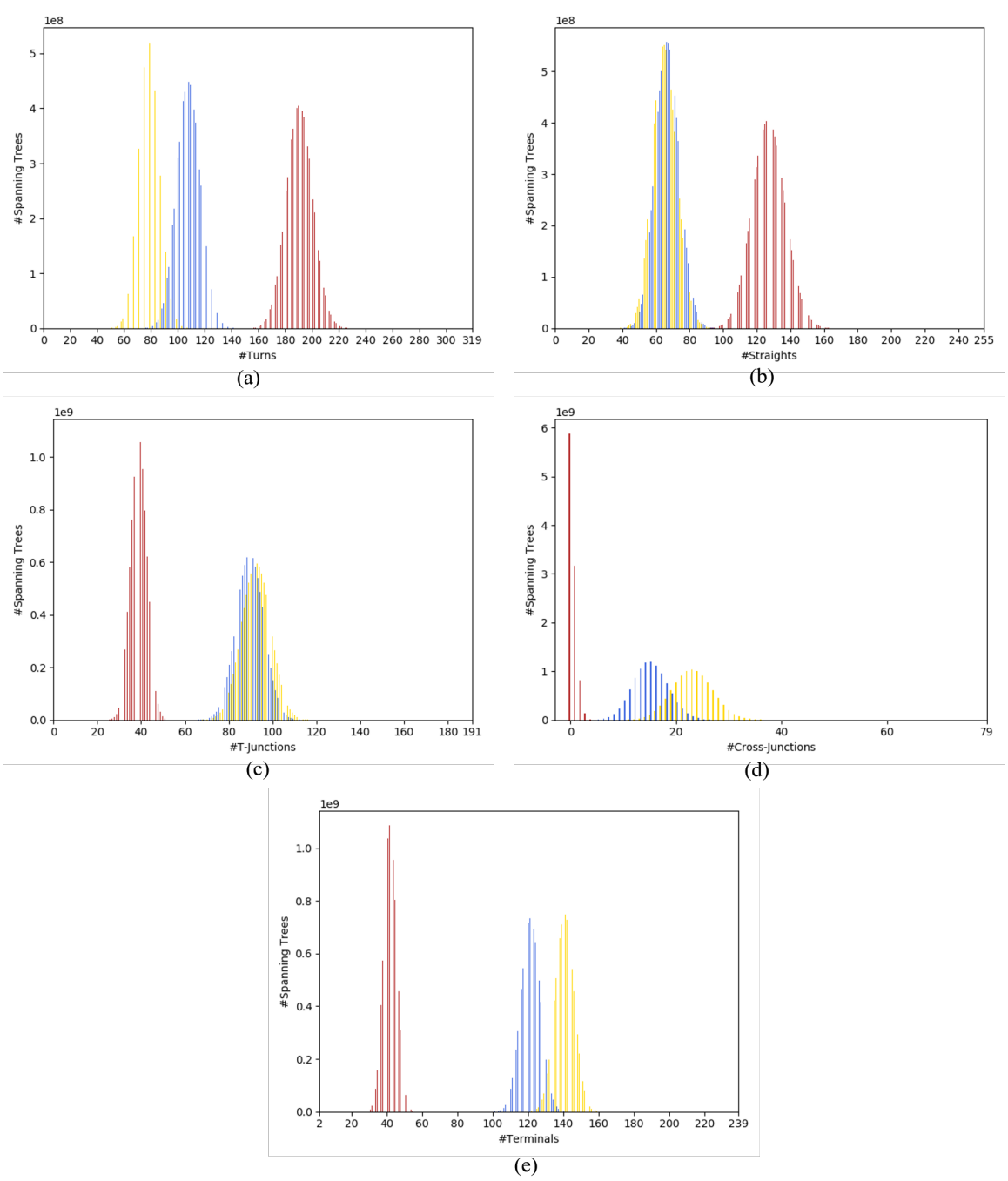


Figure 11.7: Histograms that show how many spanning trees sampled by each algorithm have corresponding metric values ((a)#Turns, (b)#Straights, (c)#T-Junctions, (d)#Cross-Junctions, (e)#Terminals) in a 20x20 grid.

histograms of a 6x6 grid, histograms of all algorithms were overlapped on some metric values. However, in histograms of 20x20 grid, histogram of the recursive backtracker and histograms of Prims and Kruskals algorithms are hardly overlapped. It means that signatures of these algorithms are shown more apparently in 20x20 grid compared to 6x6 grid. Also, in a 20x20 grid, when we compare Prims algorithm and Kruskals algorithm, the trend, in which Prims algorithm has metric values leaning to the end of a range more than Kruskals algorithm, is shown more apparently than a 6x6 grid.

As shown in Figure Figures 11.8, 11.9, and 11.10, like a 6x6 grid, Prims and Kruskals algorithms have similar shape and location of area of sampled spanning trees compared to the area of the recursive backtracker. But, compared to 6x6 grid, Prims area of sampled spanning trees leans to the boundary of a space of two metrics more than Kruskals one in 20x20 grid. For example, in Figure 11.8(b), Prims area is more closed to the left boundary of a space of two metrics than Kruskals area. In Figure 11.10(a), all algorithms have a bumpy texture like in a 6x6 grid.

Overall, sampled spanning trees of the recursive backtracker likely have higher values on #Turns and #Straights than Prim's and Kruskal's algorithms, which denote that spanning trees with long windy passages and less branches are generated easier by the recursive backtracker than the other algorithms. Sampled spanning trees of Prims algorithm and Kruskals algorithm likely have higher values on #T-Junctions, #Cross-Junctions, and #Terminals than the recursive backtracker, which denote spanning trees with lots of branches are generated easier by Prim's and Kruskal's algorithms than the recursive backtracker. When we compare Prims algorithm and Kruskals algorithm, Kruskals algorithm has a little bit weaker tendency to have many branches and less turns on sampled spanning trees than Prims algorithm. These findings indicate that different algorithms show different signatures over

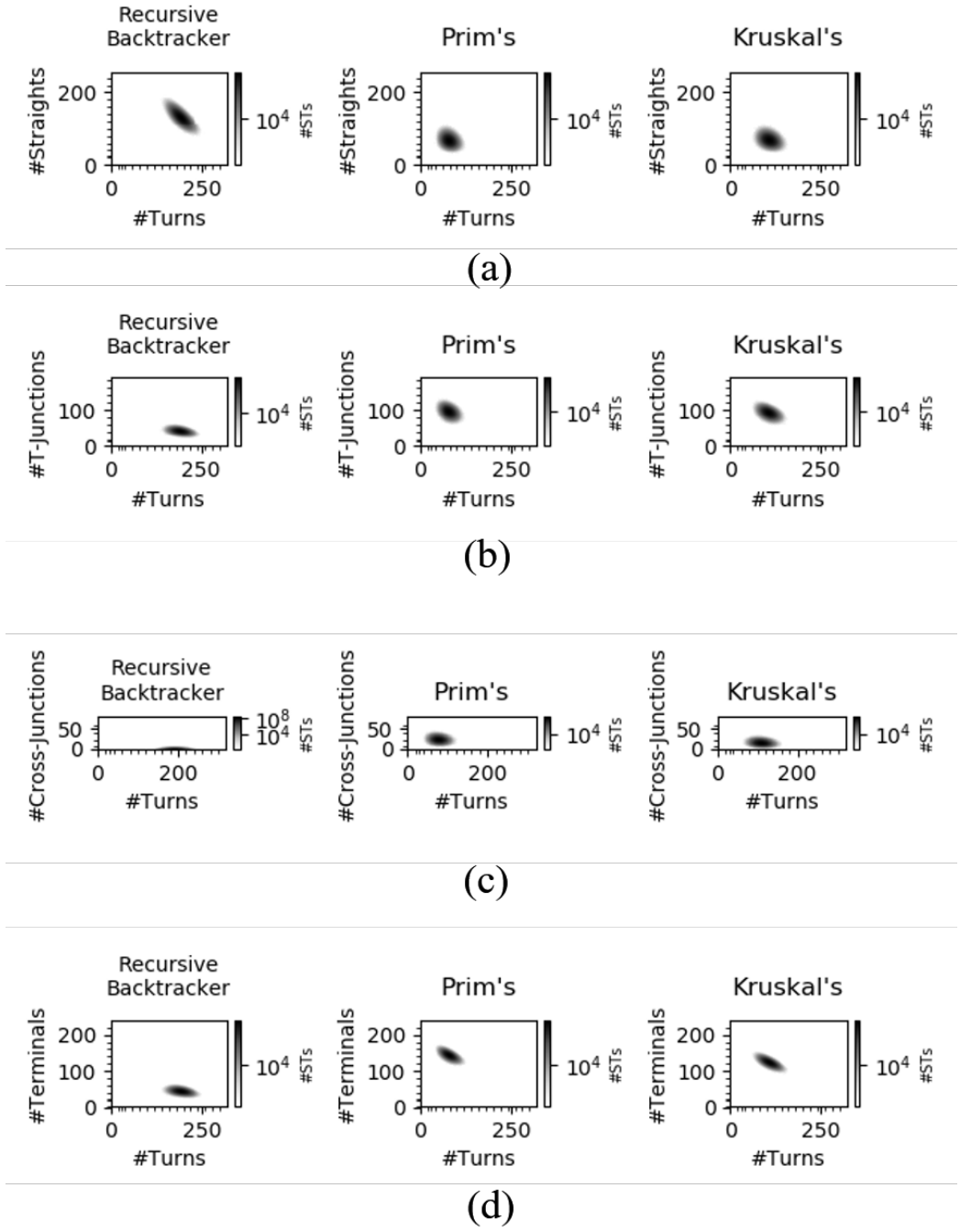


Figure 11.8: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#Turns & #Straights, (b)#Turns & #T-Junctions, (c)#Turns & #Cross-Junctions, (d)#Turns & #Terminals) by generation algorithms in a 20x20 grid. Note that values on the colorbar are in a logarithmic scale.

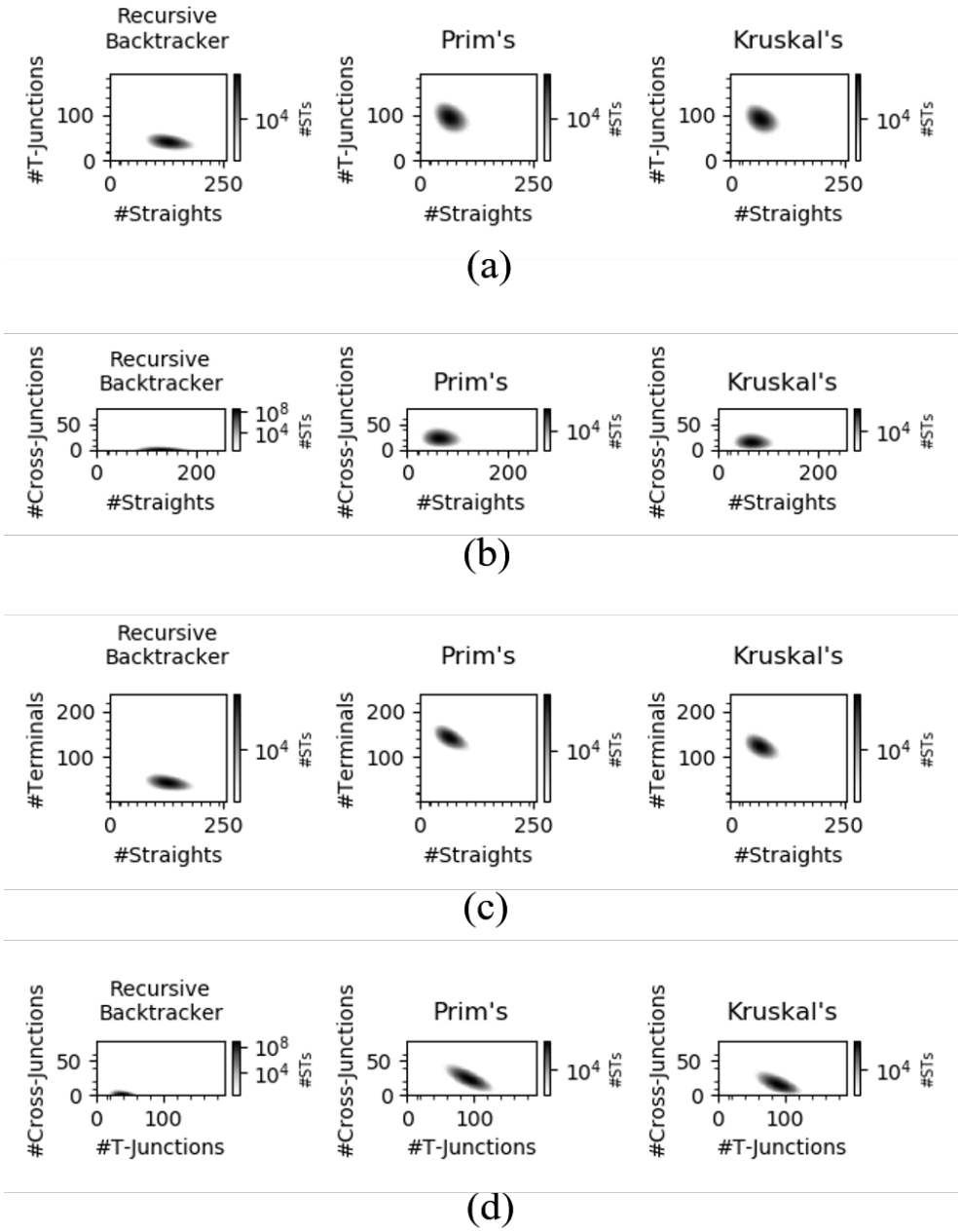


Figure 11.9: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#Straights & #T-Junctions, (b)#Straights & #Cross-Junctions, (c)#Straights & #Terminals, (d)#T-Junctions & #Cross-Junctions) by generation algorithms in a 20x20 size. Note that values on the colorbar are in a logarithmic scale.

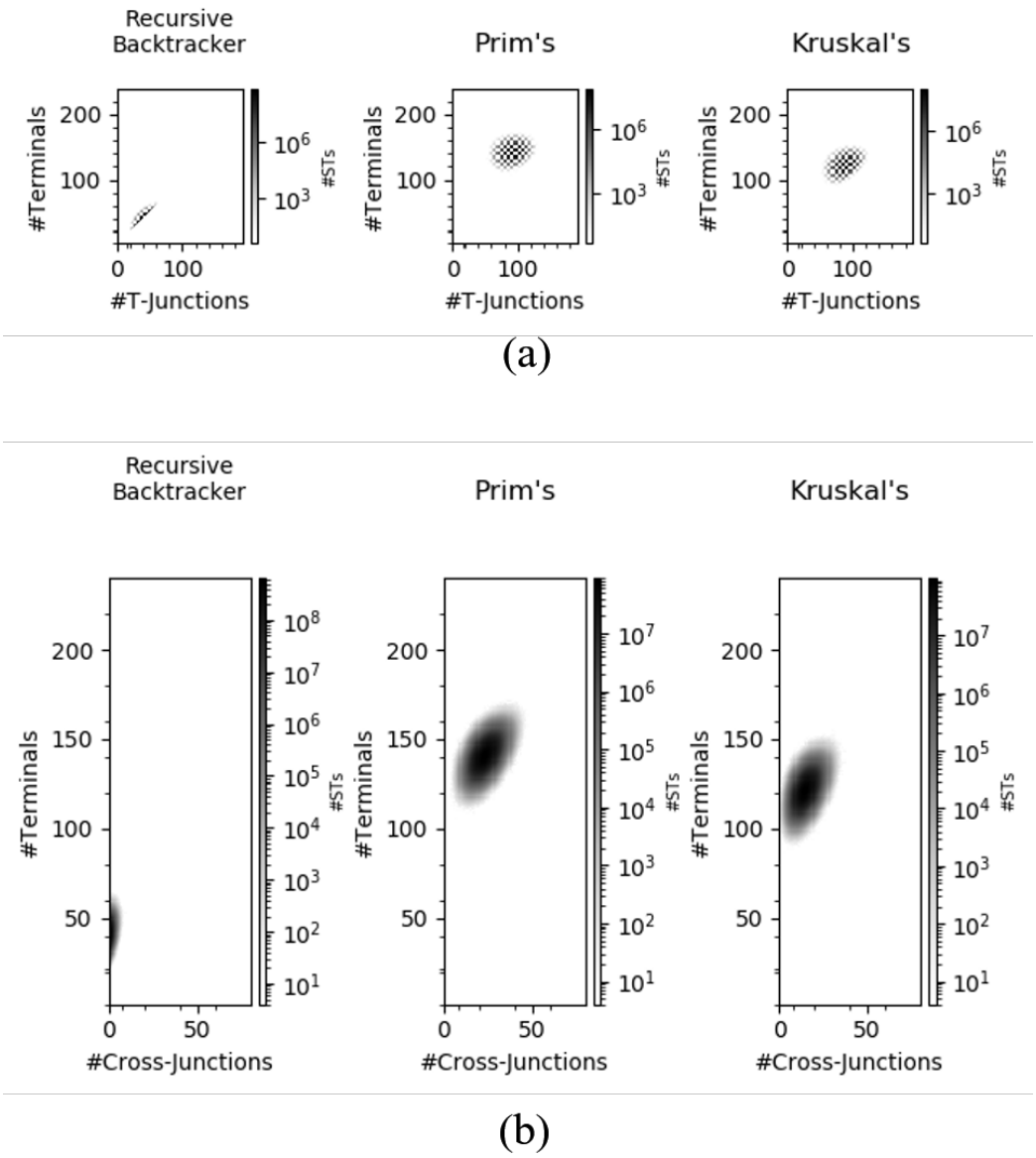


Figure 11.10: 2D histograms that show how many spanning trees have corresponding 2D combination of metric values ((a)#T-Junctions & #Terminals, (b)#Cross-Junctions & #Terminals) by generation algorithms in a 20x20 size. Note that values on the colorbar are in a logarithmic scale.

the resulting spanning trees. They can miss some spanning trees but will be very good at finding their preferred spanning trees. For example, spanning trees with many turns can be found very hard by uniform spanning tree sampling algorithms but much easier by the recursive backtracker.

In SBPCG approach, we can use one of these algorithms to find desired spanning trees. However, since each algorithm has its biased aspect on sampled spanning trees, if we choose sampling algorithm randomly amongst and use it in SBPCG approach, some desired spanning trees will be found with very low probability. For example, when desired spanning trees are spanning trees with lots of branches, if we use the recursive backtracker in SBPCG approach, the probability of having the desired spanning trees will be very low. Reversely, when desired spanning trees are spanning trees with many turns, if we use either Prims algorithm or Kruskal algorithm in SBPCG approach, the probability of having the desired spanning trees will be very low. To address this limitation of using these algorithms in SBPCG approach, in the next chapter, we suggest other SBPCG-based method, which can give us a higher probability to find desired spanning trees using these algorithms.

Chapter 12: Maze Generation Method with Choices of Algorithm among Classical Algorithms

When desired properties of a maze are given, to find the desired maze, we can iteratively generate mazes using one of existing maze generation algorithm in SBPCG until the satisfactory one is found. However, in Chapter 11, we analyzed mazes (spanning trees) sampled by spanning tree generation algorithms (the recursive backtracker, Prims algorithm, and Kruskal's algorithm) and found some biased aspect of the algorithms. From these analyses, we could see that each algorithm has tendency on its generated mazes. For example, the recursive backtracker tends to generate mazes with long windy passages and less branches. But, on the contrary, the recursive backtracker is hard to generate mazes with lots of branches. Thus, if we choose one algorithm randomly amongst and use it in SBPCG approach to generate desired mazes, some desired mazes can be very hard to find. In this chapter, we introduce a SBPCG-based method which chooses appropriate algorithm amongst based on given desired properties.

12.1 Maze Generation in SBPCG

When mazes are generated in the SBPCG approach, existing maze generation algorithms can be used. In SBPCG, we can use any single algorithm or use them all together. In this

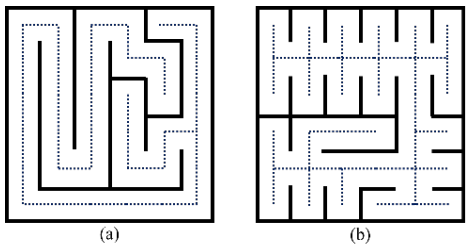


Figure 12.1: Mazes generated easily by the recursive backtracker (a) and Prims algorithm (b). Dotted lines are used to show a maze topology apparently.

section, we investigate possible approaches to apply the algorithms in the SBPCG approach and which approach will find the desired maze efficiently.

12.1.1 Using Any Single Algorithm

We can choose any algorithm at random to find desired mazes in SBPCG approach. However, as described in Chapter 11, they have different characteristics over resulting mazes. As shown in Figure 12.1, the maze with long straight ways can be generated easily by the recursive backtracker, and the maze with many branches can be generated easily by Prims algorithm. Thus, if we use any single algorithm in SBPCG approach, the probability to find some desired maze will depend on which algorithm is used for generation.

12.1.2 Using All Algorithms at Once

To address the limitation of using a single algorithm, we can use all algorithms at once in SBPCG approach. However, running all the algorithms can also be a waste of time. Lets assume that we run the recursive backtracker first and Prims algorithm next in a maze generation process. When the desired properties are given, if the corresponding maze is very hard to be generated by the recursive backtracker, it can take a very long time. In this case, it would be better to choose and use Prims algorithm right away.

12.1.3 Choosing the Best Algorithm

In our research, we choose the best algorithm based on the given desired properties and use it in the SBPCG approach. In the next section, we explain how to choose the best maze generation algorithm amongst using the sampling spaces of the algorithms.

12.2 Choice of The Best Maze Generation Algorithm

To choose the best algorithm, we construct histograms from each algorithm. If one algorithm is very good at generating mazes with some specific properties, a histogram of the algorithm will show a large number of the corresponding mazes.

12.2.1 Histograms of Metrics

Histogram of metrics can be used. A histogram of metrics shows the number of mazes for each value of the corresponding metric. When we have a histogram of #Terminals, for each value of #Terminals, we can see how many mazes are generated by the corresponding algorithm.

Having histograms precomputed, we calculate a probability of the corresponding algorithm to have a desired maze using equation (12.1).

$$\frac{\#MZ_{MetricValues}}{\#Total\ Samples} \tag{12.1}$$

In Equation (12.1), $\#Total\ Samples$ represents the total number of mazes in a histogram, and $\#MZ_{MetricValues}$ represents the number of mazes that have $MetricValues$ in a histogram. $MetricValues$ can be either single metric value or multiple metric values. Lets assume that we desire 15 turns on a maze. A histogram of #Turns is obtained from 100 mazes sampled by one of the algorithms. When there are 10 mazes that have 15 turns in the histogram ($\#MZ_{\#Turns=15} = 10$), the probability of the corresponding algorithm to

Metric Vector *mv*	**#Mazes with *mv***
(29, 20, 20, 3, 28)	412
(28, 20, 22, 2, 28)	467
(23, 20, 26, 1, 30)	117
(28, 19, 18, 5, 30)	150

Table 12.1: Example of the histogram of metric vectors. In this table, metric vector *mv* consists of metrics of maze topology (#Turns, #Straights, #T-Junctions, #Cross-Junctions, #Terminals).

have the desired maze is $\frac{10}{100} = 0.1$ by Equation (12.1). For each algorithm, the probability of having a desired maze is calculated, and the algorithm with the highest probability is chosen as the best algorithm.

12.2.2 Histogram of Metric Vectors

Here, we consider using a histogram of metric vectors to find the best algorithm. The metric vector consists of values of all maze metrics, (#Turns, #Straights,..., SolutionPath-Length), and a histogram of metric vectors shows the number of mazes found for each metric vector. An example of a histogram of metric vectors is shown in Table 12.1. In this example, instead of showing all metrics in a metric vector, we show metrics of maze topology, (#Turns, #Straights, #T-Junctions,#Cross-Junctions, #Terminals), and the number of mazes with the corresponding metric vector.

To calculate the probability of having a desired maze using a histogram of metric vectors, we can use equation (12.1). When desired metric values, $MetricValues$, are given, we find metric vectors *mv* that have $MetricValues$ in the histogram of metric vectors. Next, we add up the number of mazes of each *mv* to calculate $\#MZ_{MetricValues}$ of Equation (12.1). Then, with the total number mazes in the histogram, $\#Total\ Samples$, the probability is computed. For example, when we compute the probability of having a maze

with 30 terminals using Table 12.1, since the last two metric vectors have 30 terminals, $\#MZ_{\#Terminals=30}$ is 267 = 117 + 115. $\#Total\ Samples$ is 1146 = 412 + 467 + 117 + 150, and the probability is computed as $0.23 = \frac{267}{1146}$.

This approach pre-generates and stores one histogram of metric vectors for each algorithm while the other approach pre-computes and stores multiple histograms of metrics for each algorithm. When the probabilities of the algorithms are computed with histograms of metric vectors, the algorithm with the highest probability is chosen and used in SBPCG approach.

12.3 Performance Demonstration

In this section, we provide several use cases and demonstrate that our SBPCG-based method generates a desired maze effectively by choosing the best algorithm for each use case. For use cases, instead of setting specific metric values, we set the desired amount of metrics in percentage. For example, when we want a maze to be fully filled with turns, we input 100% for #Turns.

In our method, we use a histogram of metric vectors to choose the best algorithm. To construct the histograms of all the algorithms, we sampled 1,000,000,000 mazes using each algorithm in a 10x10 grid. When we choose the best algorithm, if there is a case that all the algorithms do not have a maze with desired metric value in their histograms, our method uses the next closest value that can be found in the histograms.

In the SBPCG approach, the process generates 10,000 mazes, and top 10 mazes, which are close to desired properties, are stored. The average value of distances of the stored mazes to desired properties is calculated and used to represent the performance of the algorithm. Tables 12.2 and 12.3 show the performance of all maze generation algorithms

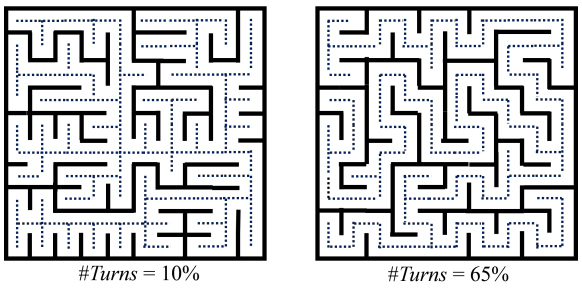


Figure 12.2: Resulted mazes based on the different desired amount of turns. Dotted lines are used to represent a maze topology.

and our method for each use case. By bolding the numbers with italic style in Tables 12.2 and 12.3, we can see that which algorithm is chosen as the best algorithm by our method to find a desired maze.

Case 1. In this case, we input different values for #Turns metric (10%, 65%) and see whether our method uses the best algorithm to generate the corresponding maze. In Table 12.2, we can see that, for #Turns=10%, our method chooses the Prims algorithm, which has the smallest average distance to desired property (0.0). For #Turns=65%, the recursive backtracker, which shows outstanding performance compared to other algorithms, is chosen as the best algorithm. In Figure 12.2, for each desired amount of #Turns, the best maze generated by our method is shown.

Case 2. In this case, we input different desired amounts of #Terminals (10%, 50%). In Table 12.2, for #Terminals=10%, the recursive backtracker with a value of 0.0 is chosen by our method to find the maze. For #Terminals=50%, our method chooses Prims algorithm as the best algorithm which has the minimum value, 6.5, against other algorithms. In Figure 12.3, the resulted mazes, which are the best ones for the corresponding amount of #Terminals, are shown.

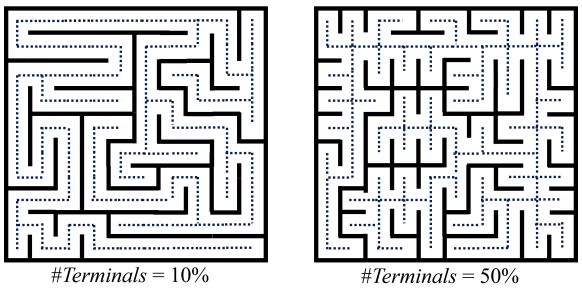


Figure 12.3: Resulted mazes based on the different desired amount of terminals. Dotted lines are used to represent a maze topology.

Case 3. In this case, we set up desired metric values of a maze for actual computer game. We want almost half space of a maze to have the solution path where players can encounter many enemies while proceeding to the end point. The rest part of the maze will be dead-end trees where users can explore to find items such as keys or treasures. In this maze, we assume the most left-upper location and the most right-bottom location as the start point and the end point, respectively. We set 35% as the desired length of the solution path (SPLength). To have different textures for the dead-end trees, we apply 40% for #TurnsDE, 35% for #StraightsDE, and 35% for #TerminalsDE, respectively. Here, these dead-end tree metrics gives a total value of corresponding metrics that all dead-end trees on a maze have. Table 12.3 shows that for each desired property, our method chooses the best algorithm to have the desired maze. Figure 12.4 represents the resulted mazes. In Figure 12.4, to distinguish the solution path and dead-end trees easily, we use dotted lines with different colors. We can see that the resulted mazes have different textures on dead-end trees corresponding to the given desired property.

Case 4. In this case, we show that users can insert topological constraints directly with some desired metric values to design a maze puzzle effectively. They can insert desired

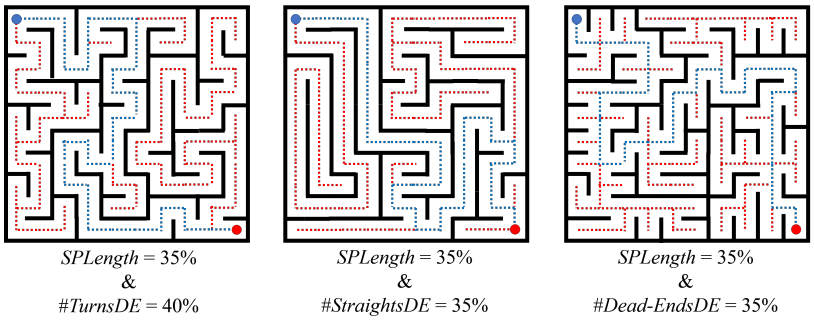


Figure 12.4: Resulted mazes based on the corresponding desired properties. In the mazes, blue dotted lines represent the solution path, and red dotted lines represent dead-end trees.

solution path directly with the start and end points and also paths of a partial area of a maze. Lets assume that we want players to get lost their way at the beginning of their play. So, when we insert desired solution path with the start and end points on a grid, we install forward dead-ends near the start point. As talked in Chapter 3, forward dead-end gives the illusion as it leads to the end point. So, to install the forward dead-ends, we insert starting branches from the solution path toward the end point and several terminal cells near the end point as shown in Figure 12.5(a). By drawing only one edge on a node with x marks, we can have a dead-end cell on the node as shown in Figure 12.5(a). Figure 12.5(b) shows a maze generated by our method with the desired properties in Figure 12.5(a). As denoted by red walls, the walls are built on the x marks to have terminal cells. Also, as denoted by blue lines, the desired solution path with partial paths are embedded on the maze.

In this chapter, we provided our method where existing maze generation algorithms are utilized with SBPCG approach to generate desired mazes. Our method chooses the best algorithm amongst using histograms of the algorithms, and then the chosen algorithm is used in SBPCG approach to generate desired mazes effectively.

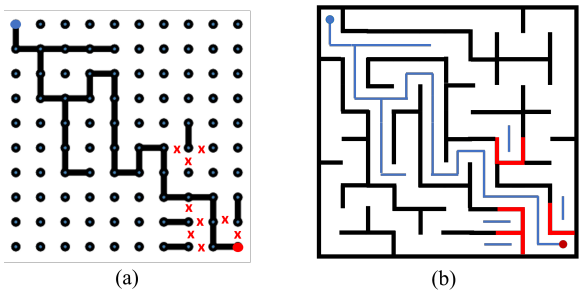


Figure 12.5: (a) User-specified topological constraints on a grid. (b) Maze generated with the constraints in (a). Blue lines and red walls in (b) represent that the maze contains the given user constraints

In this method, existing maze generation algorithms are not guaranteed to generate all possible maze topologies on a grid. In Chapter 10, by observing an enumerated space of a maze, we could see that there are very few labyrinths, which have only two terminal cells on their topologies. It indicates that the labyrinth will have very low probability to be found by any algorithm. Figure 12.6 is a histogram of #Terminals from Chapter 11, and in this histogram, we can easily check that no algorithm is good at generating mazes with #Terminals=2, which is a labyrinth. In the next chapter, we introduce our new maze generation method which can generate any (valid) desired mazes effectively.

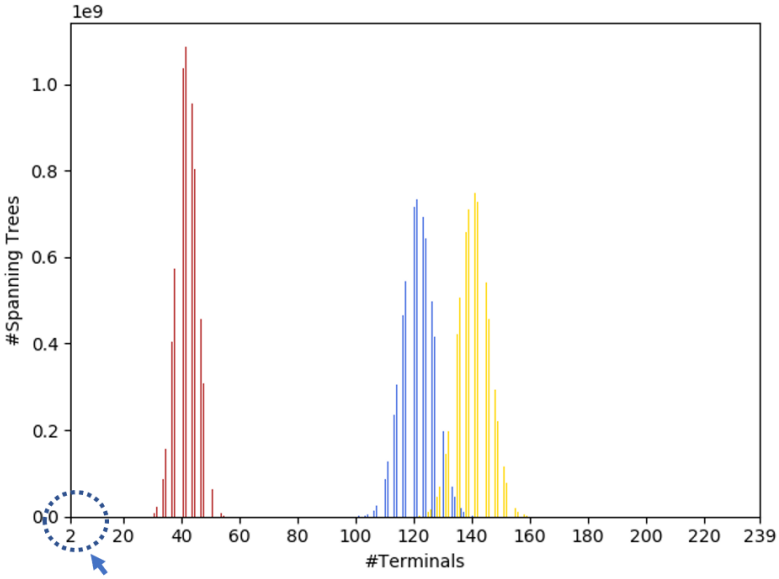


Figure 12.6: Histogram of #Terminals in a 20x20 grid from Chapter 11. Red bars, yellow bars, and blue bars correspond to the recursive backtracker, prim's, and Kruskal's, respectively. Dotted circle shows that no algorithm is good at generating mazes with two terminals (labyrinth).

	#Turns		**#Terminals**	
	10%	**65%**	**10%**	**50%**
Recursive Backtracker	20	***2.9***	***0.0***	31.9
Prim's	***0.0***	31.1	15.9	***6.5***
Kruskal's	3.1	22.9	11.6	10.6
Binary Tree	1.8	29.6	7.9	16.8
Sidewinder	2.1	26.8	9.2	13.5
Aldous-Broder	4.1	21.5	10.7	11.0
Wilson	3.2	22.4	10.2	11.9
Hunt-and-Kill	2.9	14.9	2.5	13.4
Growing Tree	7.3	15.8	5.3	16.3
Eller	1.8	27.1	9.0	13.9
Recursive Division	2.4	18.7	8.0	12
Our Method	***0.0***	***2.9***	***0.0***	***6.5***

Table 12.2: Performance of maze generation algorithms and our method based on the corresponding desired properties (#Turns=10%, #Turns=65%, #Terminals=10%, and #Terminals=50%).

	SPLength (35%)		
	#TurnsDE (40%)	**#StraightsDE (35%)**	**#TerminalsDE (35%)**
Recursive Backtracker	***1.7***	***2.7***	18.1
Prim's	17.7	16.5	4.4
Kruskal's	12.1	14.7	***1.0***
Binary Tree	17.4	16.0	16.2
Sidewinder	14.6	10.8	6.0
Aldous-Broder	11.4	13.8	2.4
Wilson	10.6	14.1	2.2
Hunt-and-Kill	8.4	8.5	2.5
Growing Tree	8.3	9.9	6.1
Eller	14.9	11.4	5.0
Recursive Division	9.1	7.4	2.8
Our Method	***1.7***	***2.7***	***1.0***

Table 12.3: Performance of maze generation algorithms and our method based on the corresponding desired properties (SPLength=35% & #TurnsDE=40%, SPLength=35% & #StraightsDE=35%, and SPLength=35% & #TerminalsDE=35%).

Chapter 13: Intelligent Maze Generation Method

To find mazes satisfying given topological properties, we iteratively generate spanning trees in a SBPCG approach. There are several ways to generate spanning trees for this search-based process, namely the many existing maze generation algorithms. However, as illustrated in Chapter 12, some algorithms have different characteristics in the created spanning trees. For example, the recursive backtracker tends to generate spanning trees with long straight ways. Thus, instead of using any specific spanning tree algorithm randomly, we choose the best algorithm for generating spanning trees in SBPCG, using our analysis from Chapter 12. But, as mentioned in Chapter 12, there are some spanning trees that are unlikely to be generated using the three existing algorithms. Therefore, that method will not be able to provide a proper result for specific sets of given topological properties.

To have 100 percent certainty that the best maze is found, each possible spanning tree must be examined by an algorithm individually. Enumerating all spanning trees guarantees that we find a solution eventually. However, the enumerated space contains a tremendously large number of spanning trees. For example, in a 6x6 grid, there are 32 trillion spanning trees as shown in Chapter 8. Whenever desired properties are given for a 6x6 grid, we need to search over the space of 4 trillion of distinct spanning trees every time. Although distinct spanning tree allows for 8 times fast speed to enumerate all spanning trees, the enumeration is still not feasible, and we want to speed up the searching process.

Table 13.1: Range of basic spanning tree metrics in a 6x6 grid.

Metric	**Min**	**Max**
#Turns	0	30
#Straights	0	24
#T-Junctions	0	16
#Cross-Junctions	0	8
#Terminals	2	22

In this chapter, we explain our new method which can generate mazes fitting any given topological properties with reasonable efficiency and accuracy.

13.1 Searching with Metric Vectors in Enumerated Space

In Chapter 10, when we analyzed the 4 trillion enumerated spanning trees for a 6x6 maze, we found out that a 6x6 enumerated space has only 1,273 metric vectors, where a metric vector consists of basic spanning tree metrics #Turns, #Straights, #T-Junctions, #Cross-Junctions, and #Terminals. The actual number of metric vectors is much less than the number of all possible spanning trees and still very small compared to the possible number of metric vectors within Table 13.1. Table 13.1 shows the range of the basic metrics. The possible number of metric vectors in Table 13.1 is 41,515, and the number 1,273 corresponds to only 3% of this. The possible number of metric vectors is computed by calculating the number of all possible combinations of metric values in a 6x6 grid, where #Turns + #Straights + #T-Junctions + #Cross-Junctions + #Terminals = 36.

This was a surprising result in our research. It indicates that the enumerated space of spanning trees for 6x6 grid could be binned into 1,273 groups using our metrics. Each group has only spanning trees with one metric vector (ex. (9,17,4,0,6)) and has that vector as its tag. Then, by looking over these 1,273 metric vectors, it may facilitate the searching

process in the enumerated space. To find a spanning tree with a specific set of topological properties, we start searching process with these 1,273 groups first. In the next section, we will describe an overview of our new method, which utilizes this metric vector space to find a desired maze.

13.2 Metric Vector-based Maze Generation

Here, we provide an overview of our new maze generator. Our goal is to search for a maze or a set of mazes satisfying a given metric vector MV_D. When MV_D is used as input, we first check that the input exists in the metric vector space. If it does not exist, we find the metric vector closest to MV_D over the possible 1,273 metric vectors. Finding the closest metric vector MV_c will be explained in Section 13.3 with more details. When MV_c is found, we reconstruct spanning trees from MV_c. Spanning tree reconstruction from MV_c will be discussed in Section 13.4. During the reconstruction, we find desired mazes satisfying MV_D or MV_c, depending on whether MV_D is within the metric vector space or not. In the following sections, we will talk about how we find MV_c among 1,273 metric vectors and how we retrieve spanning trees from MV_c.

13.3 Finding MV_c

When a user enters MV_D, it may not exist in the actual search space for the spanning tree. For example, when MV_D has #T-Junctions=3 and #Terminals=2, the MV_D cannot exist in the search space because #Terminals=2 indicates a spanning tree with no junctions. Thus, a metric vector closest to MV_D, which is MV_c, must be found among all possible metric vectors. In this section, we provide a way to find MV_c. We also discuss how to find MV_c when only a partial metric vector is used as input, such as a single metric value input.

Table 13.2: List of Metric Vectors and their Distances to MV_D=(2,11,6,3,14).

Metric Vector	**Distance to MV_D**
(7,13,7,0,9)	8
(9,17,4,0,6)	12.7
(3,10,6,3,14)	1.4
(0,10,6,4,16)	3.2

13.3.1 Calculating Distance between Metric Vectors

To find MV_c, we calculate the distance between metric vectors and find the metric vector that has the minimum distance to MV_D. Here is an example of finding MV_c in a 6x6 grid. Suppose that we have input $MV_D = (2, 11, 6, 3, 14)$. Table 13.2 has a list of possible metric vectors, and their distance to MV_D. Since the metric vector (3,10,6,3,14) has the minimum distance of 1.4 among the options, this metric vector could be chosen as MV_c.

When we have MV_D as input, to find MV_c, we can calculate the distance between metric vectors. But what if a user has a single metric value or any partial metric vector as the input? Next, we show how we manage this case.

13.3.2 Projection of Metric Vector Space

Assume that we have a single metric value as the desired topological constraint. Since this metric value is 1D, and our metric vector is 5D, we need to match the dimensions. Thus, we project our 5D metric vector space to 1D space. Then, we calculate distance between the desired metric value and the projected metric value. When the closest metric value is found in the projected space, we return the corresponding metric vectors in the original space as MV_c.

Here, we explain the projection process in more detail. In the projection process, when we project a metric vector MV into 1D space, the corresponding metric value of MV remains,

and all other metric values of MV are discarded. If there are metric vectors that have the same value of the corresponding metric, they are projected onto a single point in the 1D space. For example, when our desired property is #Straights=13, metric vectors with the same value of #Straights are projected onto one metric value in the 1D space. Figure 13.1 shows how metric vectors of Table 13.1 are projected onto 1D space of #Straights. In Figure 13.1, we can see that metric vectors with #Straights=10 are projected onto a metric value #Straights=10 as a result of projection. After the projection, we have a list of single metric values as shown in Figure 13.1. Then, to find MV_c, we compute distance between the desired metric value and the metric value in the projected space. Since our desired property is #Straights=13, the projected metric value #Straights=13 is selected, and the corresponding metric vector (7,13,7,0,9) is returned as MV_c. There is also a case that more than two metric vectors are considered as MV_c. For example, when desired property is #Straights=10, Figure 13.1 shows that we have two metric vectors (3,10,6,3,14) and (0,10,6,4,16) for MV_c. In this case, we can randomly select one as MV_c. Or, we can select all of them as MV_cs and use them one by one to reconstruct spanning trees later. For example, we can reconstruct ten spanning trees from one MV_c and ten spanning trees from another MV_c.

Likewise, when our desired property is a partial metric vector, we can apply the same projection process. Assume that our desired properties are #Straights = 10 and #T-Junctions = 6. Then, in the projection, metric vectors with the same vector of #Straights and #T-Junctions are projected onto the same vector of #Straights and #T-Junctions as shown in Figure 13.2. Since metric vectors (3,10,6,3,14) and (0,10,6,4,16) are projected onto a vector of #Straights = 10 and #T-Junctions = 6, which has minimum distance to the desired properties, the vectors are selected as MV_cs.

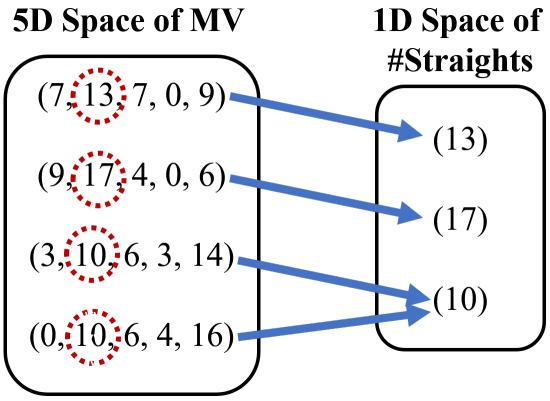


Figure 13.1: Figure that showing projection of metric vectors of Table 13.2 onto the 1D space of a metric #Straights. Since metric vectors (3,10,6,3,14) and (0,10,6,4,16) have the same value of #Straights, they are projected onto the same metric value, #Straights=10.

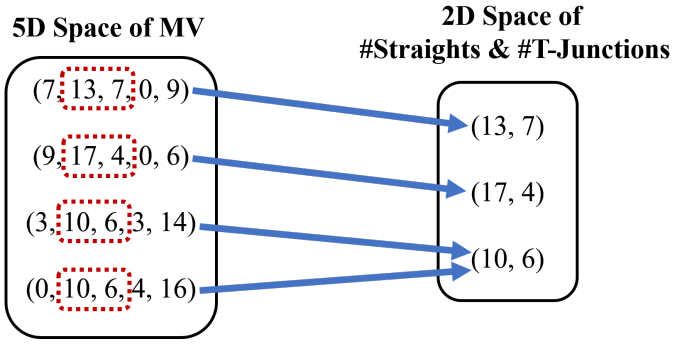


Figure 13.2: Figure that showing projection of metric vectors of Table 13.2 onto the 2D space of metrics #Straights and #T-Junctions. Since metric vectors (3,10,6,3,14) and (0,10,6,4,16) have the same vector of #Straights and #T-Junctions, they are projected onto the same 2D vector, #Straights=10 & #T-Junctions=6.

After finding one or more MV_cs, our method needs to retrieve/reconstruct spanning trees satisfying these. In the next section, we describe several ways to procedurally generating spanning trees fitting the metric vector MV_c.

13.4 Spanning Tree Reconstruction from MV_c

In this section, we provide several possible ways to recover spanning trees from MV_c and the limitations of each approach.

13.4.1 Storing All Possible Spanning Trees

First is the simplest approach to reconstructing spanning trees from MV_c. For each metric vector, MV, in a metric vector space, we pre-store a set of all possible spanning trees satisfying MV. We will denote the set of spanning trees satisfying the metric vector MV as STs_{MV}. Then, after MV_c is found, we can retrieve several spanning trees from the stored STs_{MV_c}. In this way, we are always guaranteed to have desired mazes. However, it requires storing each possible spanning tree in the enumerated space in memory, which becomes increasingly difficult and inefficient as a grid size gets larger, even with our compact bit representation. Note that we will use the notation STs_{MV} continuously in the rest of this chapter.

13.4.2 Storing Single Spanning Tree

Instead of storing all possible spanning trees, we can store one spanning tree for each metric vector MV in a metric vector space. Thus, each STs_{MV} contains a single spanning tree. Now, we need to pre-store only 1,273 spanning trees for a 6x6 grid, so it is feasible to store all STs_{MV}s. However, when we reconstruct spanning trees from MV_c, it will always yield the single spanning tree of STs_{MV_c} as a result.

13.4.3 Storing Several Spanning Trees

For each STs_{MV}, we can also store several spanning trees. Then, when we reconstruct spanning trees, we will have a more varied and interesting set of results. However, as dicussed later in this Chapter (Section 16.2), one of abilities we want in our maze generation is to directly specify paths over a maze. If we return the stored spanning trees in the spanning tree reconstruction, because all spanning tree edges are already fixed on a grid, there is no room for specifying edges directly over results. Thus, applying the hard constraints on a grid is impossible with this approach.

The above suggested methods have limitations for the spanning tree reconstruction. Therefore, in our research, we've developed an intelligent method which requires a small amount of storage yet yields spanning trees satisfying given desired properties with diverse topologies and support for user-specified paths. In the next section, we provide detailed descriptions of our spanning tree generation method.

13.5 Intelligent Spanning Tree Reconstruction

First, we represent an overview of how our method reconstructs spanning trees from a given MV_c.

13.5.1 Overview

Assume that for each metric vector MV, we have a set of spanning trees satisfying MV, which is STs_{MV}. Then, instead of storing the set STs_{MV} for MV, we analyze and construct a representation of the set STs_{MV}, which is called the representative vector. Since each MV will contain one representative vector, we only need to store 1,273 representative vectors for a 6x6 maze. Later, when we need spanning trees satisfying MV, we use the corresponding representative vector and reconstruct spanning trees from that vector. In the following

sections, we will explain our representative vector, how we construct this representation from a set of spanning trees, and how we regenerate spanning trees from the vector.

13.5.2 Representative Vector

Like the Edge Bit Vector (EBV) explained in Chapter 6, a representative vector is a vector that shows information about the edges of a grid. Each slot of a representative vector corresponds to each edge in the same manner as an EBV. In an MxN grid, a representative vector contains $(M-1)*N+(N-1)*M$ slots in total.

Each slot in our representation encodes a probability for the edge, e, to be included in STs_{MV}. Thus, when e has a value of 1 in a representative vector, it indicates that all spanning trees of STs_{MV} include this edge. Likewise, when e has a value of 0, it indicates that no spanning tree in STs_{MV} has this edge. If e has a value of 0.5, it means that half of the spanning trees in STs_{MV} have this edge.

13.5.3 Representative Vector Construction

In this section, we show how we construct a representative vector for MV from STs_{MV}. A representative vector has $(M-1)*N+(N-1)*M$ entries for a MxN grid, where each entry corresponds to a specific edge, e, for that MxN grid. To build a representative vector RV, for each edge e of RV, we compute a probability of e to be included from the set of spanning trees STs_{MV}. For the probability computation, we can use Equation (13.1) below.

$$\frac{\|\{ST \in STs_{MV} | e \in ST.Edges\}\|}{\|STs_{MV}\|} \tag{13.1}$$

In Equation (13.1), ST denotes a spanning tree, and $ST.Edges$ denotes a set of spanning tree edges. To calculate the probability value of e, we divide the number of spanning trees having e in STs_{MV} by the number of spanning trees in STs_{MV}.

13.5.4 Spanning Trees Retrieval with Representative Vectors

In this section, we show how we can recover spanning trees of STs_MV from the corresponding representative vector RV. As explained in Section 13.5.2, each entry of RV corresponds to the probability of an edge e being included in spanning trees of STs_{MV}. Then, by including edges with higher probabilities, we expect to have more probability to find spanning trees of STs_{MV}. To find the corresponding spanning trees with high probability, we need a method that includes e with higher probability more and e with lower probability less in spanning tree generation. Note that each RV focuses on finding specific spanning trees of STs_{MV}, so, using all constructed RVs, we can cover spanning trees with all possible topologies.

Minimum Spanning Tree Algorithm

To include edges with higher probability more often, we consider RV as a vector of edge weights and apply minimum spanning tree algorithms (MST) [18] over the edge weights. But, because MST chooses an edge with the lowest weight first, we modify MST in such a way that an edge with the highest weight is chosen first. In other words, we use a maximum spanning tree algorithm.

When the MST algorithm reconstructs spanning trees from RV, if there are no duplicated edge weights, a unique tree will be created. We can store and use it. If we want different topologies, we can rotate and/or mirror the store spanning tree. But, if we want a set of spanning trees beside of symmetric spanning trees, an additional action can be taken.

To prevent the result from being a single spanning tree, we modify the edge probabilities in RV.

Modifying Edge Probabilities in RV

To avoid having a single spanning tree result from RV, we modify edge probabilities in RV when we generate spanning trees from RV using MST algorithm. However, we should not assign totally random values for edges because it can break important features of RV for having desired spanning trees. For example, edges with a value of 1 or 0 indicate strong features in RV, so we need to retain those edges even after assigning new values. Thus, for each edge in RV, we allow a different range for new edge values based on its original value. Pseudocode for this new edge value computation is given in Algorithm 8. As shown in Algorithm 8, when an edge value x in RV has a maximum value of 1 or a minimum value of 0, the RV is very strong and a new x doesn't need to be computed. But, when x has a value closer to a middle value of 0.5, x has a wider range for the new value. For example, when an edge has a value of 0.5 in RV, it can have the new value between 0 and 1. This means that these edges with probabilities near 0.5 can move up toward 1, or down toward 0. Thus, by using Algorithm 8, we can have different combinations of edge weights. When we apply the MST algorithm on the different combinations, because sometimes different combinations can yield duplicates, it will not guarantee to have different spanning trees every time. But we can have much higher chance to have different spanning trees than the case of not modifying edge probabilities in RV.

13.6 Optimization of RV

As explained in Section 13.5.4, we can use RV to reconstruct spanning trees of STs_{MV}. However, when we use this RV, we can have poor performance on finding desired mazes.

Algorithm 8 Recalculate an edge's value on a mask

```
1: function RecaluclateEdgeValue(EdgeValues) ▷ EdgeValues: set of edge values in RV
2:     c ← 0.5
3:     for i ← 0 to (M − 1) ∗ (N − 1) ∗ 2 do
4:         x ← EdgeValues[i]
5:         α ← c − |x − c|
6:         maxValue ← MIN(1, x + α)
7:         minValue ← MAX(0, x − α)
8:         EdgeValues[i] ← RandomGen(minValue, maxValue)
9:                         ▷ RandomGen(a, b): returns a random number between a and b
10:    end for
11: return EdgeValues
12: end function
```

There can be a case, in which RV does not have strong features for having spanning trees of STs_{MV}, such as edge weights close to 0 or 1. In this case, most edge weights of RV are close to 0.5. When we apply the MST algorithm to reconstruct spanning trees from the RV, the RV will yields random spanning trees but not spanning trees close to desired metrics. The reason we have RV without strong features is that spanning trees of STs_{MV} do not share similar structures much over their topologies. We provide Figures 13.3 and 13.4 to show relationship between spanning tree topologies and RV. In Figures 13.3 and 13.4, edge weights of RV are represented by grayscale colors on a grid. When an edge weight is close to 1, the corresponding edge has darker color on a grid. Likewise, When an edge weight is close to 0, the corresponding edge has brighter color on a grid. As shown in Figure 13.3, when spanning trees have similar structures in their topologies, we can have RV with strong features. The RV has edges with dark color as well as bright color. But, when we have spanning trees with totally different topologies as shown in Figure 13.4, the resulting RV does not have strong features and has grey edges. It denotes that the too much diversity of STs_{MV} can act as noise during RV construction. Therefore, instead of directly

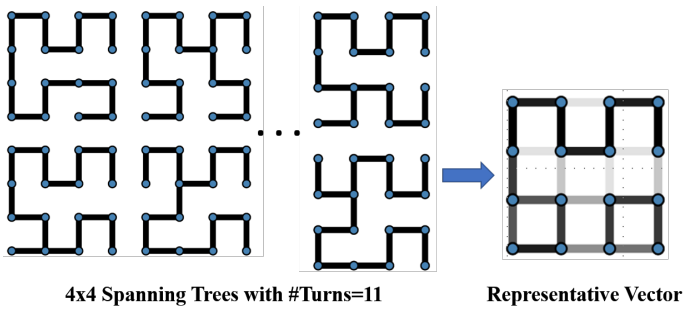


Figure 13.3: Figure that showing a set of spanning trees with #Turns=11 in a 4x4 grid and a RV constructed from the set. Edge weights of RV are represented by edges with grayscale color on a grid. When an edge weight is close to 0, the corresponding edge has brighter color on a grid. Likewise, when an edge weight is close to 1, the corresponding edge has darker color on a grid.

constructing RV from STs_{MV}, we need to optimize RV. In this optimization approach, we iterate to improve STs_{MV} with reduced noise and construct RV from the STs_{MV} until our objective function is minimized.

In the next sections, first we explain our objective function in detail. Then, we explain how we optimize RVs using the objective function.

13.6.1 Objective Function in Optimization Process

In the optimization process of RV, we use the objective function in Equation 13.2. In this section, we provide detailed description of this objective function. In the explanation, we will denote a set of spanning trees as S.

$$f = \mathrm{DIFF}_{avg} + (\alpha - UT) \tag{13.2}$$

First, we explain the term $DIFF_{avg}$ of the function in detail. DIFF_{avg} is calculated on S using (13.3).

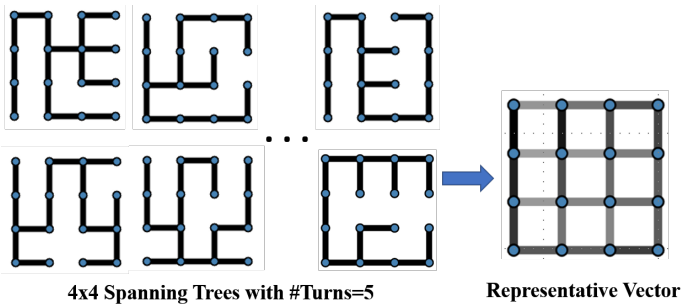


Figure 13.4: Figure that showing a set of spanning trees with #Turns=5 in a 4x4 grid and a RV constructed from the set. Like Figure 13.3, edge weights of RV are represented by edges with grayscale color on a grid.

$$\text{DIFF}_{avg} = \frac{\sum_{i=1}^{N} ||MV - V_i||}{N} \tag{13.3}$$

In Equation (13.3), N is the number of spanning trees in S, and V_i is the metric vector of one spanning tree i of S. In Equation (13.3), DIFF_{avg} is an average value of differences between V_i of spanning trees in S and the desired MV. When DIFF_{avg} is minimized, we obtain our RV.

When DIFF_{avg} is the only term optimized, the process may converge to some local space. This means that the resulting RV might yield unique and non-varied topologies. Our research would like to have various topologies in the resulting spanning trees. Thus, in addition to DIFF_{avg}, we added the second term $(\alpha - UT)$ in the function (13.2), where

$$(\alpha - UT) = \begin{cases} \alpha - UT, & \text{if } \alpha - UT > 0 \\ 0, & \text{otherwise} \end{cases} \tag{13.4}$$

In Equation (13.4), α is the desired number of unique spanning trees, and UT is the number of unique spanning trees in S. When S has at least α distinct spanning trees, which means $\alpha - UT \leq 0$, a value of $(\alpha - UT)$ becomes 0, which is the minimum value

that $(\alpha - UT)$ can have. But, when S has few distinct spanning trees, a value of $(\alpha - UT)$ is bigger than 0. Thus, by having the minimum value of $(\alpha - UT)$ in the optimization process, the optimized RV has a higher probability to generate spanning trees with at least α unique spanning trees.

13.6.2 Optimization Process

Detailed description of the objective function 13.2 was given in Section 13.6.1. Here, we show an overview of the optimization process to construct a better RV from improved STs_{MV}. Pseudocode for the optimization process is provided in Algorithm 9. We construct a representative vector RV using Equation (13.1) from the initial STs_{MV} which contains all spanning trees satisfying MV in the enumerated space. Then, to check whether the initial RV is constructed well, we generate some amount of spanning trees using that RV. In our research, we generate 10,000 spanning trees from RV in each iteration. Then, we evaluate f from the 10,000 generated spanning trees S. If $f \geq \beta$, we go to the next iteration. We build another STs_{MV} by selecting spanning trees satisfying MV in S and obtain another RV of the set STs_{MV}. Then, to check whether the RV was constructed well, we generate a set of spanning trees S using RV again. Then, f is evaluated from this S. If f is not less than β, we go to the next iteration again. This process continues until the f is less than β.

Algorithm 9 Optimization Process of Representative Vector Construction

1: $S \leftarrow ENUM$ ▷ ENUM: enumerated set of all possible spanning trees
2: $f \leftarrow MAXFLOAT$
3: **while** $f \geq \beta$ **do**
4: $STs_{MV} \leftarrow GenSTs_{MV}(S)$
5: ▷ $GenSTs_{MV}(S)$: returns a set of spanning trees satisfying MV in S
6: $RV \leftarrow GenRV(STs_{MV})$ ▷ $GenRV(X)$: returns RV from a set X
7: $S \leftarrow GenSTs(RV)$ ▷ $GenSTs(RV)$: returns spanning tree set constructed from RV
8: $f \leftarrow Computef(S)$ ▷ $Computef(S)$: computes f from S
9: **end while**

Figure 13.5 provides histograms to prove the optimized RV works better than original RV. Each histogram corresponds to each desired property in a 6x6 grid and show performances of original RV and optimized RV for having the desired spanning trees. To demonstrate the performance, using each RV of each desired property, we generated 1,000 spanning trees in a 6x6 grid. As shown in Figure 13.5, optimized RVs reconstruct the desired spanning trees more frequently than original RVs. Most of spanning trees generated from the optimized RVs are focused on desired metric values or near desired metric values. For example, in Figure 13.5 (a), when the desired property is #Turns=2, the histogram of original RV has the highest frequency on #Turns=6, and the histogram of optimized RV has the highest frequency on #Turns=2. In Figure 13.5 (c), when the desired property is #T-Junctions=16, the histogram of optimized RV has the highest frequency is on #T-Junctions=15 and also covers #T-Junctions=16 while the histogram of original RV does not cover #T-Junctions=16. Thus, from the histograms in Figure 13.5, we can see that maze generation can be controlled well based on the given desired properties with optimized RVs.

After RVs are optimized, we pre-store them. When desired property is given, in a SBPCG approach, we bring RV corresponding to the desired property and use it to iteratively generate spanning trees.

13.7 Performance Demonstration

Effectiveness of our method is demonstrated in this section. We demonstrate how our method generates desired mazes on target. Also, we provide some use cases to show how we generate game content for a game level satisfying this use-case.

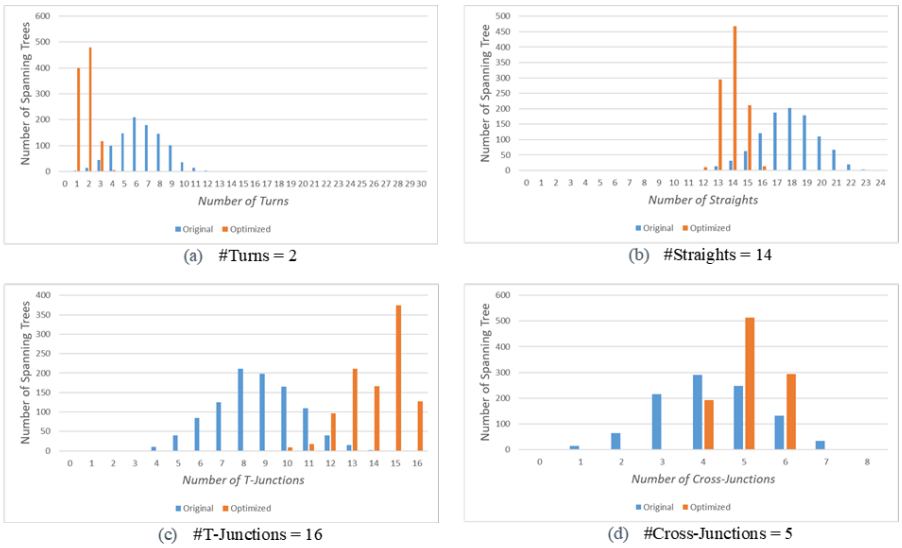


Figure 13.5: Figures that show histograms of original RVs (blue bars) and optimized RVs (orange bars) on a 6x6 grid targeted toward the metric in the caption. For each histogram, 1,000 spanning trees were generated using each RV in a 6x6 grid. We can see that optimized RVs gives us higher frequencies on given topological constraints than original RVs.

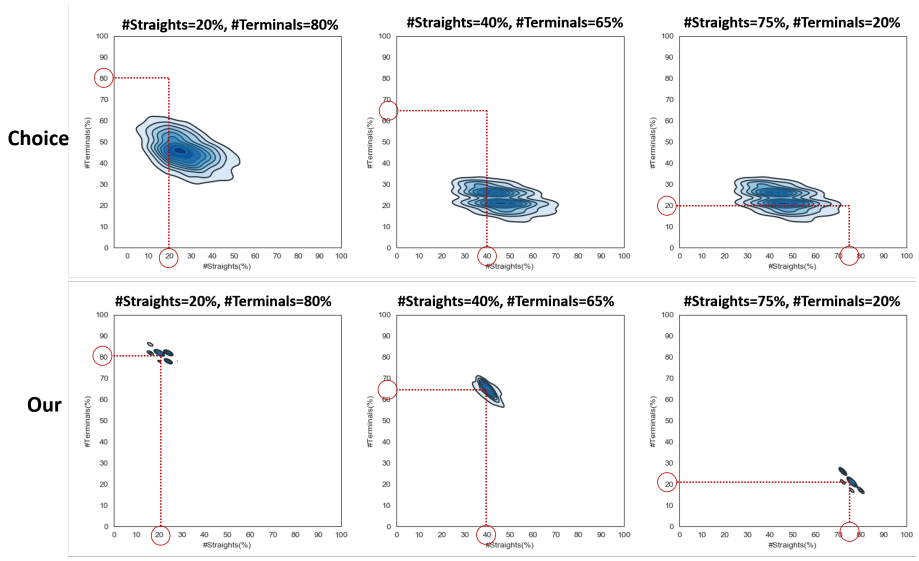
First, we measured the expressive range [35] of our new maze generator and compared it with the expressive range of the choice-based generator in Chapter 12. This selects the best algorithm among the recursive backtracker, Prim's algorithm, and Kruskal's algorithm so it denotes that we are comparing performance to those. To obtain expressive range data, we input various combinations of 2D metric values and generated 1,000 mazes on a 6x6 grid, for each combination using each generation method. In Figure 13.6, the expressive ranges are visualized using 2D histograms. In the histograms, the darkness of color corresponds to the number of mazes sampled with metrics at the location/bin. The darker color means that more mazes belong to that bin. Red-dotted lines are used to show whether expressive ranges (generative spaces) cover mazes of the corresponding metric values. Note, in this

application, we desire a small expressive range near the target, ideally only at the deisred metrics.

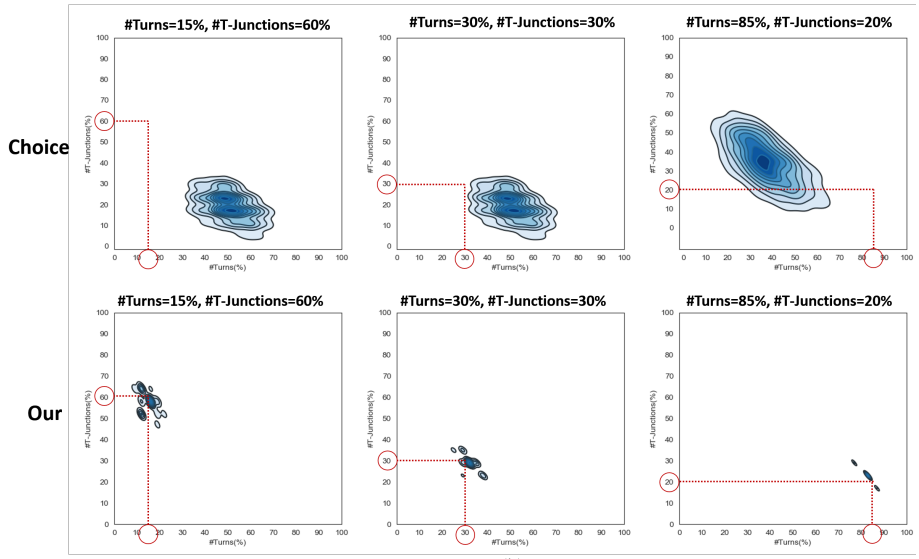
Figure 13.6 shows that we can have direct control over resulting topologies using our method. While the choice-based method does not have a big change in expressive ranges based upon input metric values, our method properly adjusts its expressive ranges toward the desired input metric value. It indicates that our method can generate desired mazes effectively because the maze generation can be controlled well by our method.

Next, we provide some use cases that reflect properties desired in designing actual computer game levels and show how we can use our maze generation method to generate the desired game levels.

Use-Case 1. Consider a use case in which a game requires a level in which the player must complete a search quest. As described in [4], the search quest is in which the player is asked to search a game level to collect required items. We might prefer that the player spend more time walking around within the level in order to acquire each quest item. A puzzle like a labyrinth would mean that all items are on the same path, and would require little "searching", so we opt for a specific maze design that avoids this. In this design, we desire a few numbers of long curving dead-ends so that a player cannot see easily what is at the end of each dead-end in the level. This desired property can motivate a player to explore dead-ends. To generate the desired game level, in our method, we set up values of a few #Terminals and low #Straights and a relatively higher value on #Turns in a 6x6 grid so that we can have long dead-ends with a curving on the level. In Figure 13.7, we show the actual game level that we designed based on a maze generated with the desired properties by our method. We can see that the level contains a few number of long dead-ends so that a player cannot see the end of dead-ends easily while a player explores these short dead-ends.



(a)



(b)

Figure 13.6: Charts showing expressive ranges in a 6x6 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 shown 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.

Use-Case 2. Assume that we are designing a game level where a player is fleeing from enemies that chase the player. To make this game level have more tension, we can design the game level to have a single path, such as a labyrinth, with a high run. Then, the player is forced to look ahead and has to deal with any events on the single path to the exit. Also, if the maze has fewer junctions, the player will feel that they have less control over evading the enemy by escaping at a fork in the road and has to rely on running as fast as possible to escape. This will build fear and anxiety into the gameplay. To generate the desired game level, we set up a minimum value on #Terminals, which is 2, and relatively higher value on #Straights in a 6x6 grid. In Figure 13.8, we can see the actual game level designed based on a maze generated by our method. We can see that the level has only one path with a gloomy atmosphere so that a player can have tension while they are fleeing from enemies.

Note that, in each use-case, we iteratively generated mazes using a SBPCG approach until the exact satisfactory maze is found. Table 13.3 shows that our method has a much faster search time compared to the choice-based method in the use-cases. Notably, the desired maze of the use-case 1 is very hard to find using the choice-based method, which takes more than 4 minutes, but is very easy to find using our method, which takes less than 0.1 seconds. It demonstrates that our new method can be effective in designing computer game content.

In this chapter, we explained how to build representative vectors to generate mazes with topological constraints. In Section 13.7, we observed that they have very good results for small grid sizes, like 6x6. However, it is not feasible to find results efficiently with a larger grid. The reason is that it is computationally prohibitive to obtain enumerated data in a larger grid where our optimization process starts. In a 6x6 grid, it takes three days to

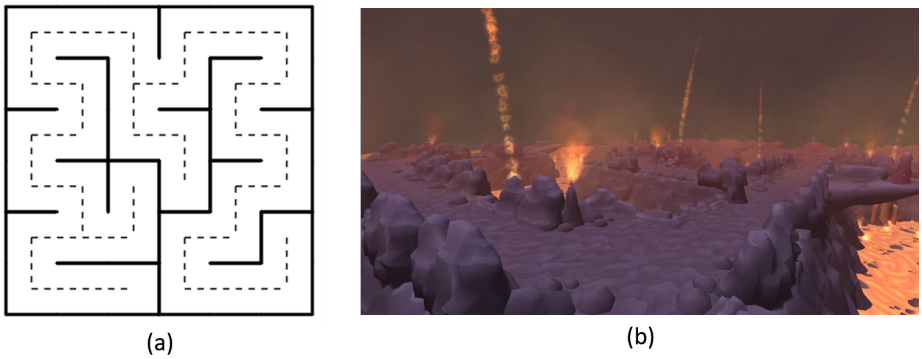


Figure 13.7: (a) Structure of the game level for Use-Case 1, which is designed based on a 6x6 maze generated by our method. (b) Actual gameplay view of (a).

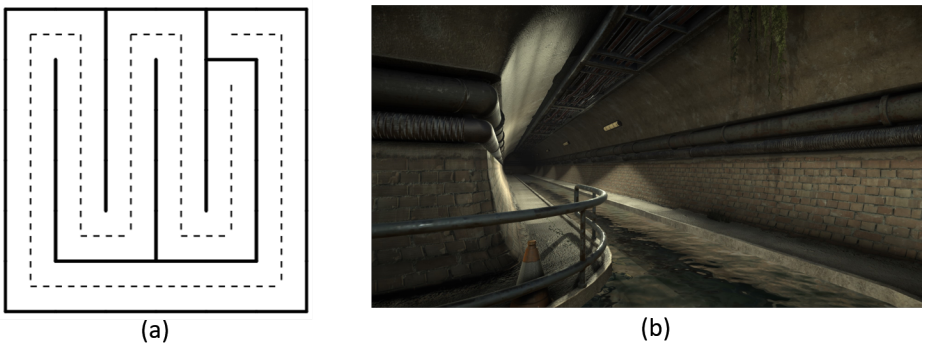


Figure 13.8: (a) Structure of the game level for Use-Case 2, which is designed based on a 6x6 maze generated by our method. (b) Actual gameplay view of (a).

Table 13.3: Table showing input metrics and time(s) to search desired maze in each use-case.

	Input Metrics	**Choice-based Method**	**Our Method**
Use-Case 1	#Turns = 90% #Straights = 5% #Terminals = 5%	274s	**0.07s**
Use-Case 2	#Straights = 90% #Terminals = 2	13s	**0.03s**

enumerate all possible spanning trees using 500 cores of a cloud computing system. In the next chapter, we explain how we generate desired mazes in larger size of grid.

Chapter 14: Large Maze Generation with Desired Properties

As discussed in the previous chapter, our representative vectors, which works well in a small grid, have a limitation in a large grid. To create a maze larger than 6x6 with desired properties, we apply a new process, the concept of a hierarchical maze. In this chapter, we explain what a hierarchical maze is and how we generate a large maze satisfying given desired properties using the hierarchical maze. Note that it works in theory but is not practical for large mazes. Many times, mazes of size 6x6 or less are useful.

14.1 Hierarchical Maze

As shown in Figure 14.1, a hierarchical maze is a maze where mazes are stitched together. Figure 14.1 shows a 12x12 hierarchical maze where four 6x6 mazes are stitched. Depending on where the mazes are stitched, we can have different sizes of a hierarchical maze. For example, using four 6x6 mazes, we can have a 12x12 maze but also a 6x24 or 24x6 maze.

14.2 Hierarchical Maze Generation with Desired Properties

To find a maze larger than 6x6, we generate hierarchical mazes with given desired properties in a SBPCG approach. In this section, we show how we generate a hierarchical maze given the desired properties.

First, when desired properties are given as metric values, we distribute the metric values over a set of small mazes. The values can be distributed either evenly or heterogeneously.

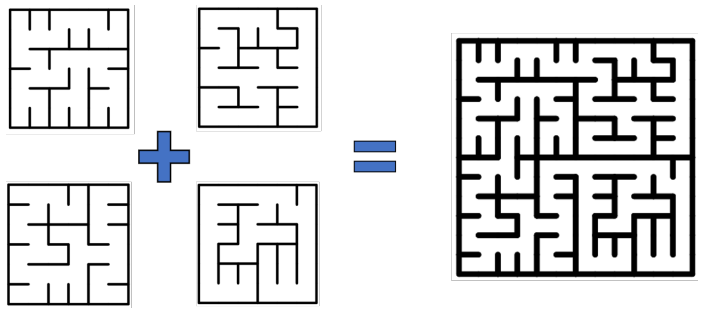


Figure 14.1: Overview of hierarchical maze construction. 6x6 small mazes are stitched to build 12x12 large maze.

For example, when we stitch four 6x6 mazes as shown in Figure 14.1, if our desired property for the large maze is #Turns = 40 in a 12x12 grid, we can divide the metric evenly so that we have #Turns = 10 for each 6x6 maze. It may give us a large maze which has the proper metric desired when the mazes are stitched.

After we set up desired metrics for the small mazes, we use representative vectors to create the small mazes with the corresponding metrics. After we generate the small mazes, we generate a hierarchical maze by stitching them together.

In the next section, we will explain this stitching process in detail.

14.3 Stitching Process

When we place four small mazes adjacent to each other as shown in Figure 14.2(a) with exterior walls, we need to knock down boundaries from these so that we have a single perfect maze as a result. As represented in Figure 14.2(a), we have four boundaries. In the stitching process, we first choose which boundaries to stitch. Then we choose which wall to knock down on each chosen boundary. In Figure 14.2(b), we can see that when we stitch two 6x6 mazes, there are six possible walls to knock down on each boundary. By knocking

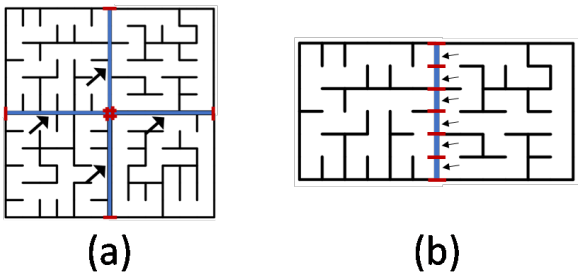


Figure 14.2: (a) Figure showing four possible boundaries to choose (denoted by arrows). (b) The Figure showing six possible walls to knock down between mazes (denoted by arrows).

down one wall at a time on the chosen boundaries, we have a perfect maze. The boundary selection step and wall selection step will be discussed in detail next.

14.3.1 Choosing Boundaries

In this boundary choosing process, we can erroneously choose a random boundary among the possible boundaries. However, this random selection can make a loop or a disconnection on a hierarchical maze as shown in Figure 14.3. This creates an invalid perfect maze. Thus, it is important to choose appropriate boundaries to avoid loops and isolated areas.

To choose proper boundaries, we construct a spanning tree graph based on the number of 6x6 small mazes that are being stitched together. For instance, if we are making a 12x12 hierarchical maze, we have four 6x6 mazes arranged 2x2. Therefore, we create a spanning tree for a 2x2 grid. Assume each small maze is a node. Then, we can convert the combined mazes to a 2x2 grid with four nodes as shown in Figure 14.4. Each edge of the grid corresponds to each boundary of the combined mazes. After we have a 2x2 grid structure from the mazes, we generate a spanning tree on that 2x2 grid, which is called a hierarchical connection tree. Then, by choosing the boundaries corresponding to the edges of the hierarchical connection tree, we will have a hierarchical maze without any loops or

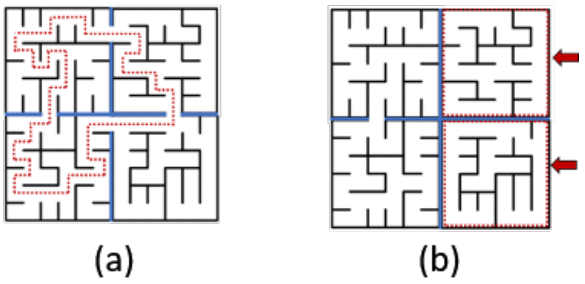


Figure 14.3: (a) Figure showing that we can have a loop when we knock down all boundaries. The loop is denoted by the red-dotted line. (b) Figure showing that we can have isolated areas denoted with arrows when we knock down a single boundary.

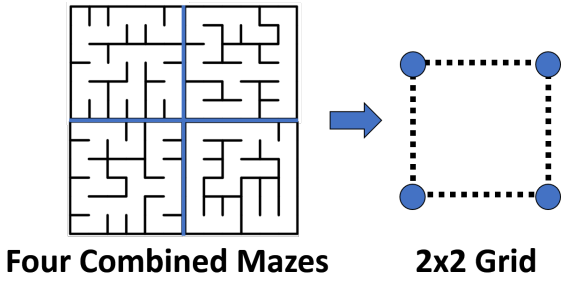


Figure 14.4: Figure showing that four combined mazes can be converted to a 2x2 grid for a hierarchical connection tree.

isolated area. To help readers understand this hierarchical connected tree-based process visually, we provide an illustration of the process in Figure 14.5.

After we choose the boundaries, we choose a specific wall to knock down on each boundary. In the next section, we will explain how to choose walls on each boundary.

14.3.2 Choosing Walls to Knock Down on Boundaries

When we choose wall for knocking down, if we choose more than one wall on a boundary, we will introduce a loop as shown in Figure 14.6. Thus, on each boundary, we choose a

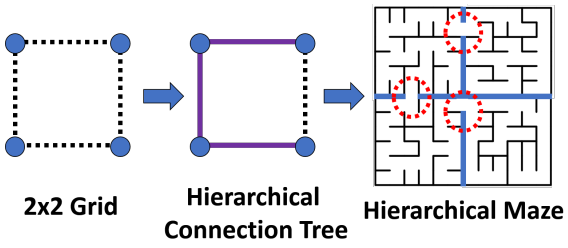


Figure 14.5: Figure showing that boundaries corresponding to edges of a hierarchical connection tree of Figure 14.4 are knocked down as denoted by the dotted circles to create a valid hierarchical maze.

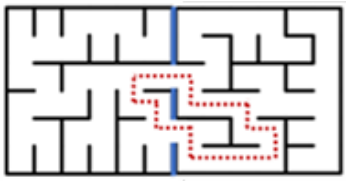


Figure 14.6: The example that has a loop (denoted by the dotted path) by knocking down more than one wall on a boundary.

single wall to knock down. We can choose any single wall randomly. However, our goal in building a hierarchical maze is to have metric values close to the desired metric value.

We will illustrate some issue with the random selection in terms of spanning trees instead of mazes. When we knock down a wall between two mazes, it will be represented as connecting two spanning trees by adding an edge between them.

When we connect spanning trees by adding an edge between them, we change the connectivity of the nodes adjacent to the edge introduced. If a node in question was a turn cell, it becomes a T-Junction cell. Likewise, Straight cells and T-Junction cells become T-Junction cells and Cross-Junction cells, respectively. If the node with the new edge is a Terminal cell, it will become a Turn cell or a Straight cell. Thus, by adding an edge, metric

values of a hierarchical maze will be changed slightly. When we have Q boundaries, the maximum distance is $(Q-1)*2$. If the adjacent nodes were a turn cell and a straight cell initially, then they will both become T-junction cells with this new edge. Then, the values of both the #Turns and #Straights of the hierarchical maze will be decreased by 1, and value of #T-Junctions of the hierarchical maze will be increased by 2.

Thus, in our wall selection step, when we check if a wall on a boundary is appropriate to remove, we first check how introducing that edge affects the metric values of the hierarchical maze. Then, we choose the set of edges that results in the metrics of the hierarchical maze being as close as possible to the desired metrics.

Here, we give an overview of how we generate desired large maze using a SBPCG approach. In the SBPCG approach, when desired metric values input, first the metric values are divided evenly or heterogeneously over small mazes. Then, for each small maze, the corresponding RV is used to generate a spanning trees satisfying the assigned metric values. After we obtain all small mazes using RVs, we construct a hierarchical connection tree over the small mazes. When we stitch the small mazes, we use the hierarchical connection tree to find boundaries between the small mazes to knock down. Then, the best wall on each boundary is knocked down to have a large desired maze. By iteratively generating large mazes in the SBPCG approach, we search for the desired maze. This hierarchical maze-based approach can limit a generative space but still yield somewhat diversity set of resulting mazes as shown in Figure 14.7.

14.4 Performance Demonstration

In this section, we demonstrate the performance of our method in finding large mazes with topological properties.

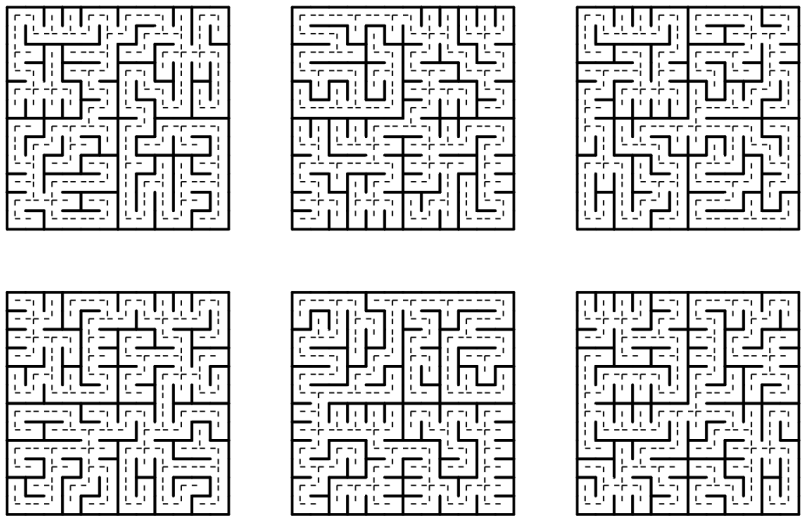


Figure 14.7: Figure showing resulting hierarchical mazes with #Turns = 50% and #T-Junctions = 30% in a 12x12 grid. The dotted-lines are used to show topologies apparently.

To demonstrate the stable performance with the stitching work, like we did in Section 13.7, we generated 1,000 mazes in an 18x18 grid and plotted expressive range data in Figure 14.8. As shown in Figure 14.8, even with the larger scale, our new method has high control over generative space compared to the choice-based method so that any possible desired mazes can be found easily in a large grid. Choice-based method have no mazes with desired metrics in some cases. But, our new method always has samples with desired metrics for these cases.

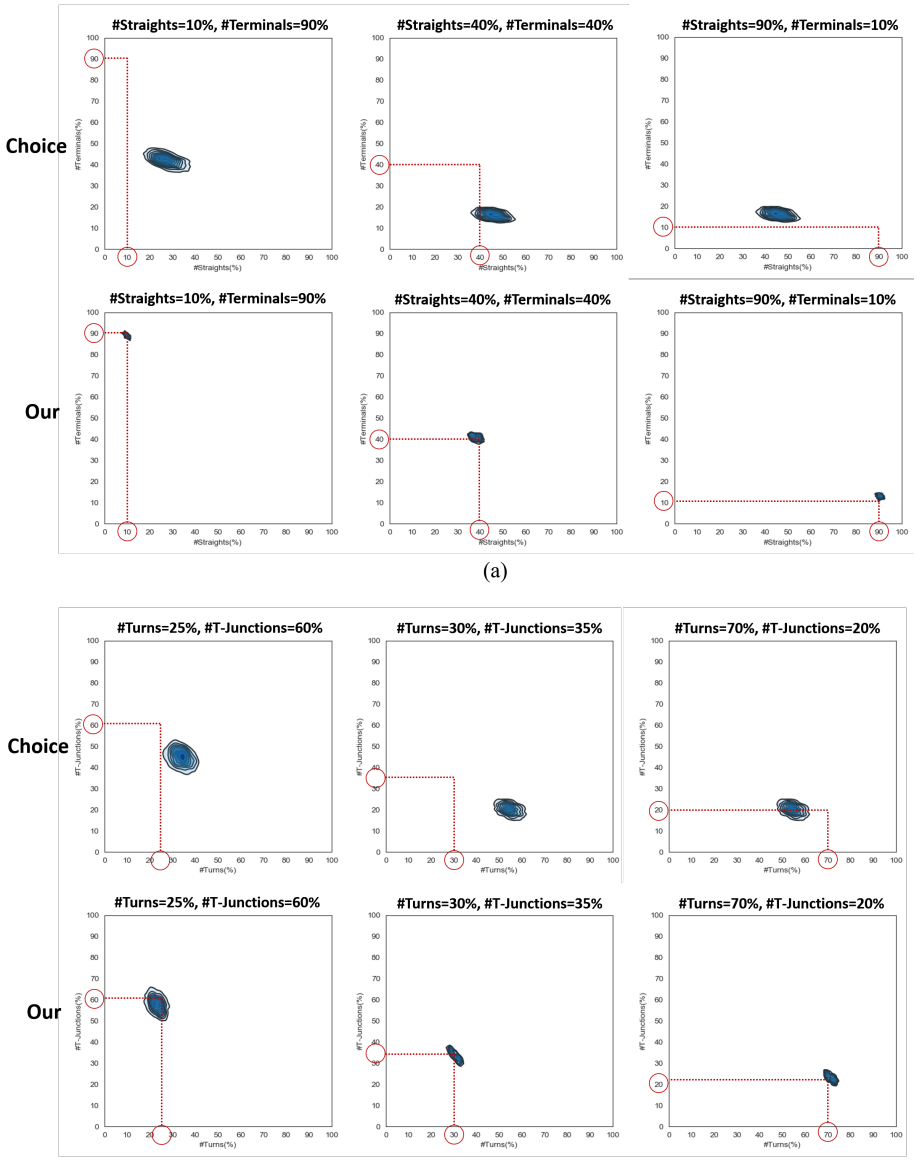


Figure 14.8: Charts showing expressive ranges in a 18x18 grid regarding 2D metrics ((a) #Straights & #Terminals, (b)#Turns & #T-Junctions).

Chapter 15: Path Generation based on Topological Constraints

When we have a maze puzzle to solve, we need to find a path from the starting point to the ending point. This path is called the solution path and is an important component on a maze. The structure of the solution path can result in different playing experiences. For example, as shown in Figure 15.1, even if we have the same structure of a maze, the difficulty level of solving the maze is different due to the different solution path. The short solution path allows the puzzle be solved easily. Also, by having many decisions on the solution path, we can make a player be disoriented to find a way to the ending point. Therefore, the solution path is important in maze design, but current existing maze research usually focuses on the maze itself and not the solution path. Thus, we provide a method that gives detailed control over the solution path topology. In our research, as explained in Chapter 17, after having desired solution path, a maze is generated from the path so that we guarantee to have the maze containing the desired solution path. In this chapter, we will explain our desired path generation method.

15.1 Domain

15.1.1 Rectangular Grid

A rectangular grid is a grid where rectangle cells are arranged. When we have a MxN grid, we have rectangle cells arranged in M columns and N rows. The center of each cell is called a node, and a MxN grid has M*N nodes in the grid. Also, line segments between

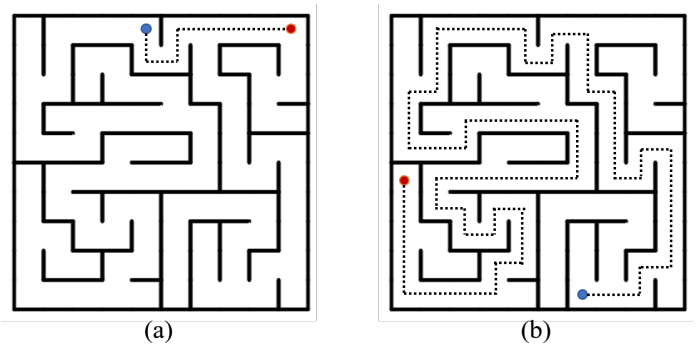


Figure 15.1: Figures that show the same maze structure with different solution paths. In both figures, blue dot and red dots denote the starting point and the ending point, respectively. A dotted line is used to show the solution path. We can see that a maze with the short solution path (a) is much less difficult to solve than a maze with a long windy solution path (b).

nodes are called edges. On a rectangular grid, each node has a neighboring edge in either vertical or horizontal direction. Thus, each node can have four neighboring edges at most, and a MxN grid has $M\,(N-1) + (M-1)\,N$ edges in total.

15.1.2 Path on Rectangular Grid

When we have a path, each end of the path is called an end point. We have two end points in a path on the solution path of a maze, the starting point and ending point. In a spanning tree, any two nodes can be the end points of a path. In our research, we set the top-left node and the bottom-right node as the two end points as default.

Consider that a node in an MxN grid, v_i, where $0 \leq i \leq MN-1$. A path of a rectangular grid can be defined as follows. Given two end points v_s and v_e, a path on a rectangular grid is a sequence of consecutive nodes $v_s, v_{s+1}, \ldots, v_e$, where v_i and v_{i+1} are adjacent either vertically or horizontally. No node appears more than once in the sequence. Hence, the path cannot backtrack or have loops.

In this chapter, when we explain our method for desired path generation, we explain it in terms of vertical and horizontal edges.

15.2 Desired Properties of Paths

In this section, we define desired properties of a path on a grid. For each desired property, path metrics can be measured. Path metrics are based on cell types.

15.2.1 Path Metrics

Here, we define some path metrics that can be used to define desired properties.

- #Turns: Denotes the number or percentage of turn cells on a path.
- #Straights: Denotes the number or percentage of straight cells on a path.
- PathLength: Denotes the number of edges of a path.
- Max#ContiguousStraightCells: Denotes the maximum number or percentage of contiguous straight cells where no turn cell exists between the straight cells on a path. This metrics can be used in a function for more design-centric property such as speed and agility.

When we define desired properties of a path, we give desired values for path metrics. In the next sections, we discuss how to generate a path with the desired properties on a grid.

15.3 Path Generation

Before talking about desired path generation, we explain how we generate a path on a grid. To generate a path on a grid, we can use several existing path generation algorithms. First, we can apply graph traverse algorithms such as randomized depth-first search (DFS) which chooses a neighboring node to visit randomly. Since the algorithm can be used for

path finding problems, it can give us paths on a grid. Each algorithm has a bias. For example, since DFS tends to move forward through neighboring nodes, we usually have a resulting path with long straight ways. Thus, there will be some paths that are very hard to find using those traversal algorithms. Alternatively, applying a random walk algorithm which results in all paths being possible with the same probability. However, sometimes it takes a very long time to generate a path and does not guarantee that the algorithm terminates. We can say that these algorithms are not appropriate if we want to generate paths with desired properties, as these give the user no control. In the next section, we introduce our method called the column-sweeping method, which can generate paths quickly and find the desired path efficiently.

15.4 Column-Sweeping Method

In our research for the path generation we use our method called column-sweeping method. In this method, we can generate a path rapidly. Also, we can apply constraints to control path generation. By applying constraints in the method, we can reduce time for finding the desired path. Here, we will explain this column-sweeping method.

15.4.1 Column

To explain column-sweeping method, we need to explain the concept of a column first. Columns denote sub-regions of a grid. Each column has corresponding vertical edges and horizontal edges as shown in Figure 15.2. For example, the leftmost column of a grid has the leftmost vertical edges and horizontal edges. Note that all columns except the rightmost column have corresponding vertical and horizontal edges, but the rightmost column has vertical edges only. In our method, we generate edges of a path on each column, and it means that edges of a path are selected among edges of the column.

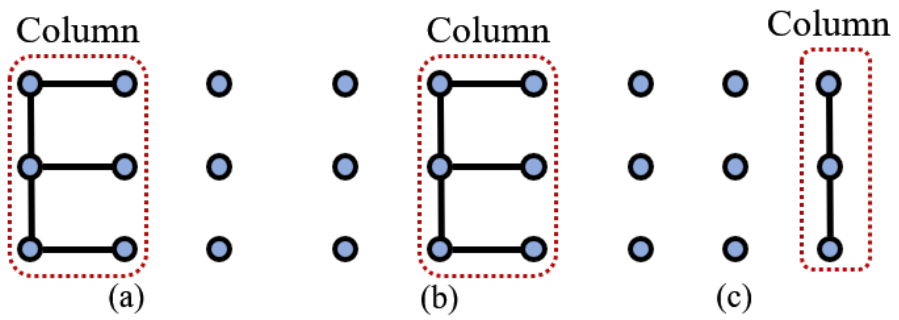


Figure 15.2: Figures show corresponding vertical and horizontal edges of each column on a 3x3 grid. (a) Edges of the leftmost column. (b) Edges of the second left column. (c) Edges of the rightmost column. Note that the rightmost column has only vertical edges.

15.4.2 Simple Column-Sweeping Method

Column sweeping method generates edges of a path based on columns. As shown in Figure 15.3, edge generation starts from the leftmost column and is swept to the rightmost column. When edges are generated on each column, we can generate edges arbitrarily. However, this arbitrary process will not guarantee that we have a valid path at the end. On each column, we need to generate edges that can give us a path later.

Here, we explain how we generate valid edges, which will give us a path, from each column. In a MxN grid, we denote the leftmost column as col_1, rightmost column as col_M, and i$^{\text{th}}$ column from the left as col_i. When we generate edges on one column col_i, we check whether there are loops or junctions between all previous columns by applying traversal algorithm over the columns. While traversing the generated edges, if it meets already visited node, we have loops. Also, if it backtracks to already visited node and moves to another unvisited node, we have junctions. If the edges on col_i make loops or junctions, we regenerate other edges on col_i and check again whether that edges make loops or junctions between col_1 and col_i. When we have valid edges on col_i, we move to col_{i+1} and generate

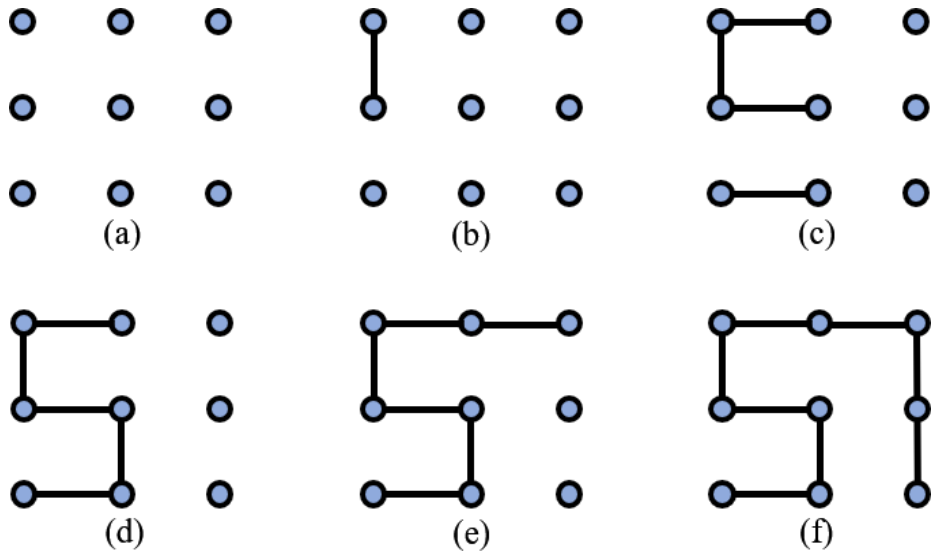


Figure 15.3: Illustration of path generation process using column sweeping technique on a 3x3 grid.

edges on col_{i+1}. Thus, after our column-sweeping process ends, if we have a single path, we store it. Otherwise, it is discarded.

This current method can give us a path as a result, but it requires exhaustive scanning work during the process. Thus, we make this current column-sweeping method more efficient by introducing some new concepts and several rules. Next, we explain our new efficient column-sweeping method.

15.4.3 Efficient Column-Sweeping Method

In this section, we introduce our new column-sweeping method that can generate a path on a grid efficiently.

First, we introduce some new concepts to explain the method.

- #NE: Each node of a grid has a value of #NE. It denotes the number of edges of a path adjacent to a node. When a node has two adjacent path edges, this node has #NE=2.

Likewise, when a node has no adjacent path edges, this node has #NE=0. This concept helps us to avoid multiple disconnected paths by preventing nodes from having #NE=1. Also, we can avoid junctions on a path using this concept by preventing nodes from having #NE=3 or #NE=4. All nodes of a grid have #NE=0 initially. For a path on a grid, all nodes of the path except the two end points have #NE=2. Because each end point has only one adjacent edge by the path, the end points become to have #NE=1. If a disconnect exists on the path, then there are at least three nodes with #NE=1. Thus, to detect the disconnect, we need to check which nodes have #NE=1 and whether the nodes are end points or not. By setting up different initial #NE values for end points, we can make this checking process easier.

Initial value of #NE of End Points: To detect disconnects on a path without checking whether nodes are end points or not, we setup initial values of #NE for end points as #NE=1. Then, when we have a path on a grid, because each end point has only one adjacent edge by the path, all nodes of the grid including the end points have either #NE=2 or #NE=0. If there are nodes with #NE=1 on a grid, we can know that the grid has disconnected paths. From now on, we assume that end points on a grid have #NE=1 initially.

- SetID: Each node of a grid has a SetID. As shown in Figure 15.4(a), when there is no path edge on a grid, all nodes of the grid have different SetIDs, which means that they belong to their own set. As shown in Figure 15.4(b), if some nodes are connected by the edges, those nodes will then have the same SetID, which means that they belong to the same set.

Figure 15.4: Small numbers denote SetIDs. (a) Grid with no edges. Every node has its own SetID. (b) Grid with some edges. Connected nodes have the same SetID.

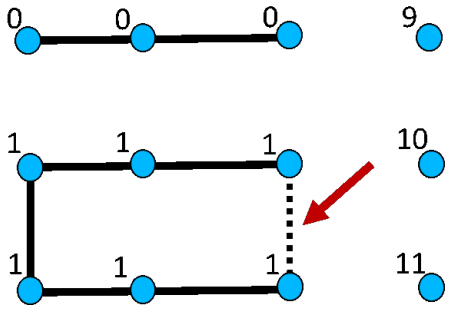


Figure 15.5: Small numbers denote SetID of corresponding nodes. If we add a vertical edge to the place marked by the red arrow, where nodes of the edge have the same SetID, it will give us a loop.

Next, we provide three rules that apply the above concepts to reduce scanning work and make the current method efficient.

Rule 1. Before a vertical edge is placed between nodes on a column of a grid, the nodes of the prospect vertical edge should have different SetIDs.

If nodes have the same SetID, it denotes that the nodes are already connected by edges on the grid. If we place a vertical edge between them, it will give us a loop as shown in Figure 15.5. Thus, a vertical edge can only be placed between nodes with different SetIDs.

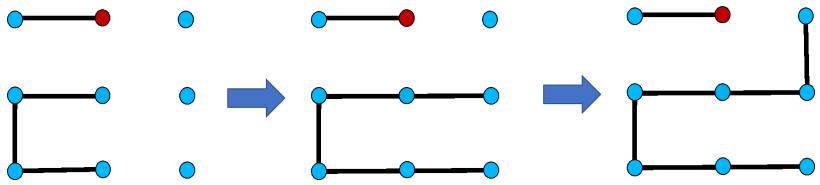


Figure 15.6: If we do not add a horizontal edge on the second column next to the red node which has #NE=1, the node will be a terminal cell later as shown in the rightmost graph.

Rule 2. When two neighboring nodes on a column of a grid have #NE < 2, then a vertical edge can be placed.

If a node has #NE=2, by adding a vertical edge to the node, the node will have #NE=3, which denotes a junction on a grid. Thus, a vertical edge should only be placed between nodes with #NE < 2.

Rule 3. Horizontal edges are required from nodes having #NE=1 after #NEs are updated from vertical edges.

On one column, if a horizontal edge is not placed next to a node which has #NE=1 on that column, the node will remain with #NE=1. As shown in Figure 15.6, the node becomes a terminal cell, and there will be a disconnect. If a horizontal edge is placed next to a node, which has #NE=0 or #NE=2, the node will have #NE=1 or #NE=3. As shown in Figure 15.7, the node becomes terminal cell or T-junction cell, which is not allowed. Thus, horizontal edges should be added next to nodes which have #NE=1 after vertical edges have been placed.

Rule 4. No horizontal edges are placed to nodes with #NE$\neq 1$.

If a horizontal edge is placed next to the node with with #NE=0, the node will be orphaned. Also, if a horizontal edge is placed next to the node with #NE ≥ 2, the node will have a junction.

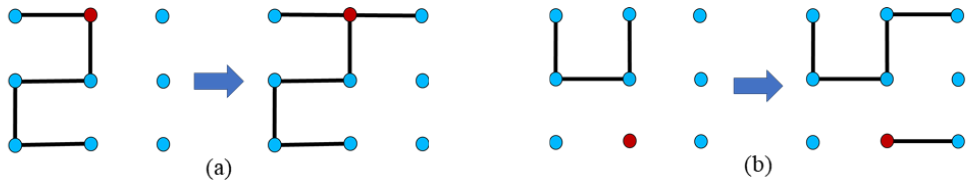


Figure 15.7: (a) Figure showing the case that a horizontal edge is added next to the red node, which has #NE=2. The node will change to have #NE=3, which means a T-junction cell, which is not desirable on a path. (b) Figure showing the case that a horizontal edge is added next to the red node, which has #NE=0. The node changes to have #NE=1, which means a terminal cell not desirable on a path.

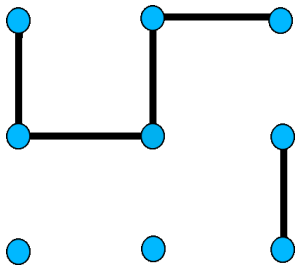


Figure 15.8: Example showing that nodes on the last third column violate Condition 1 after applying the column sweeping technique with the rules.

In this new column-sweeping method, when we generate vertical edges and horizontal edges on each col_i, we generate the edges satisfying the above the rules. By using the rules, any invalid case is prevented between col_1 to col_{M-1}.

However, after we finish this sweeping process on col_M, we need to check whether the generated edges over the grid form a single path because we can have multiple disconnected paths. When we generate vertical edges on col_M, rule 1 and rule 2 can give us nodes with #NE=1 as shown in Figure 15.8. Thus, to have a single path as the final result, we set up extra condition for the last column as shown below.

Condition 1. After column-sweeping method ends, all nodes on col_M should have either #NE=0 or #NE=2.

By preventing #NE=1 on col_M, we can prevent disconnects over a grid, and it will give us a single path as a result.

Before this column-sweeping method begins, we assign different SetIDs to all nodes of a grid as shown in Figure 15.4(a). We assign #NE=1 to two end points and #NE=0 to the rest of the nodes. Then, the column-sweeping method is processed with Rules 1, 2, 3, and 4 between the first column and the last column. After the column-sweeping ends on the last column, if a generated graph satisfies Condition 1, we store it as a path. If it is not, we throw away the graph.

Consequently, we can see that we do not need to check disconnects and junctions exhaustively during the process unlike we did in the previous simple column-sweeping method. Next, we will explain how we generate a path with desired properties using this efficient column-sweeping method.

15.5 Path Generation with Desired Properties

The algorithm above provides all possible paths between two end points. We want to find one or more paths with desired characteristics. However, when we have desired properties, it is hard to generate the desired path directly using only column-sweeping method. Instead, we apply search-based procedural content generation with our column-sweeping method.

15.5.1 Search-Based Procedural Content Generation (SBPCG)

As described in Chapter 7, SBPCG is one approach to procedurally generate the desired contents using searching mechanism. In this approach, to generate the content with desired properties, we search the desired content by generating one at a time. When we search

for the desired content, we can continue to generate content until we find the desired one. If we want the searching process to end within some amount of time, we generate a fixed number of results and find the result which has the closest metrics to the desired result. In our research, we generate paths using our column-sweeping method in SBPCG approach to find the desired paths.

We can do random generation in the generation stage of SBPCG. To randomly generate paths using column-sweeping method, on each column, we randomly generate vertical edges with Rule 1 and Rule 2. Then, horizontal edges will be determined by Rule 3. However, the random generation can continuously give us duplicate paths and take a long time to find desired paths. Instead, we enumerate all paths using our column-sweeping method. Since the enumeration generates all possible paths, it guarantees a path fitting the desired metrics is generated.

The enumeration with column-sweeping method is perhaps too computationally expensive because the enumeration problem is an NP problem, and the search space is exponentially large. In our column-sweeping method, we insert constraints based-upon the desired properties. These constraints reduce the searching time for desired paths. We explain enumeration of paths using our column-sweeping method first, including constraints.

15.5.2 Path Enumeration

Let's start our enumeration of vertical edges on col_1 with rule 1 and rule 2. On each column of an MxN grid, the possible number of vertical edge patterns is 2^{N-1}. Then, for each set of vertical edges on col_1, horizontal edges are determined by rules 3 and 4. Then, for each set of horizontal edges on col_1, we enumerate vertical edges on col_2 using Rule 1 and Rule 2. This recursion process continues until we get to the last column col_M. As explained previously, we have an extra condition for col_M. Thus, when we enumerate vertical edges

on col_M, we discard vertical edges violating Condition 1. After this enumeration process, we have created all possible paths.

In Figure 15.9, we provide an example of our enumeration process in a 3x5 grid visually so that the process becomes more understandable. Note that in this example, the top-right node and the bottom-left node are assumed as end points. We added external edges to end points so that we can see easily whether all nodes of a path have #NE=2. Also, small numbers on nodes denote SetIDs. The enumeration process starts from one pattern of edges on the first column. Next, Possible vertical edges satisfying Rule 1 and Rule 2 are generated on the second column. Then, possible horizontal edges satisfying Rule 3 and Rule 4 are generated on the second column. We can see that the horizontal edges are dictated by vertical edges on the second column. So, for each graph on the second top line in Figure 15.9, we have only one pattern of horizontal edges. After we have horizontal edges on the second column, we generate vertical edges on the last column. After finishing generating edges on the last column, checking process is performed for each generated graph. In the checking process, we check whether all nodes on the last column have either #NE=2 or #NE=0. A graph which passes the checking process is stored as a path.

15.5.3 Applying Constraints in Path Enumeration

We can reduce the enumeration step by applying some constraints. Assume that we have the desired path length as a desired property. While we do the column sweeping, before we generate edges on the current col_i, we check whether the constraints below are satisfied. If the constraints are satisfied, we generate edges on col_i. Otherwise, we generate the next enumeration of vertical and horizontal edges on the previous col_{i-1} and then move to col_i again.

Constraint 1. $E_{i-1} \leq P \ -(M-i)$

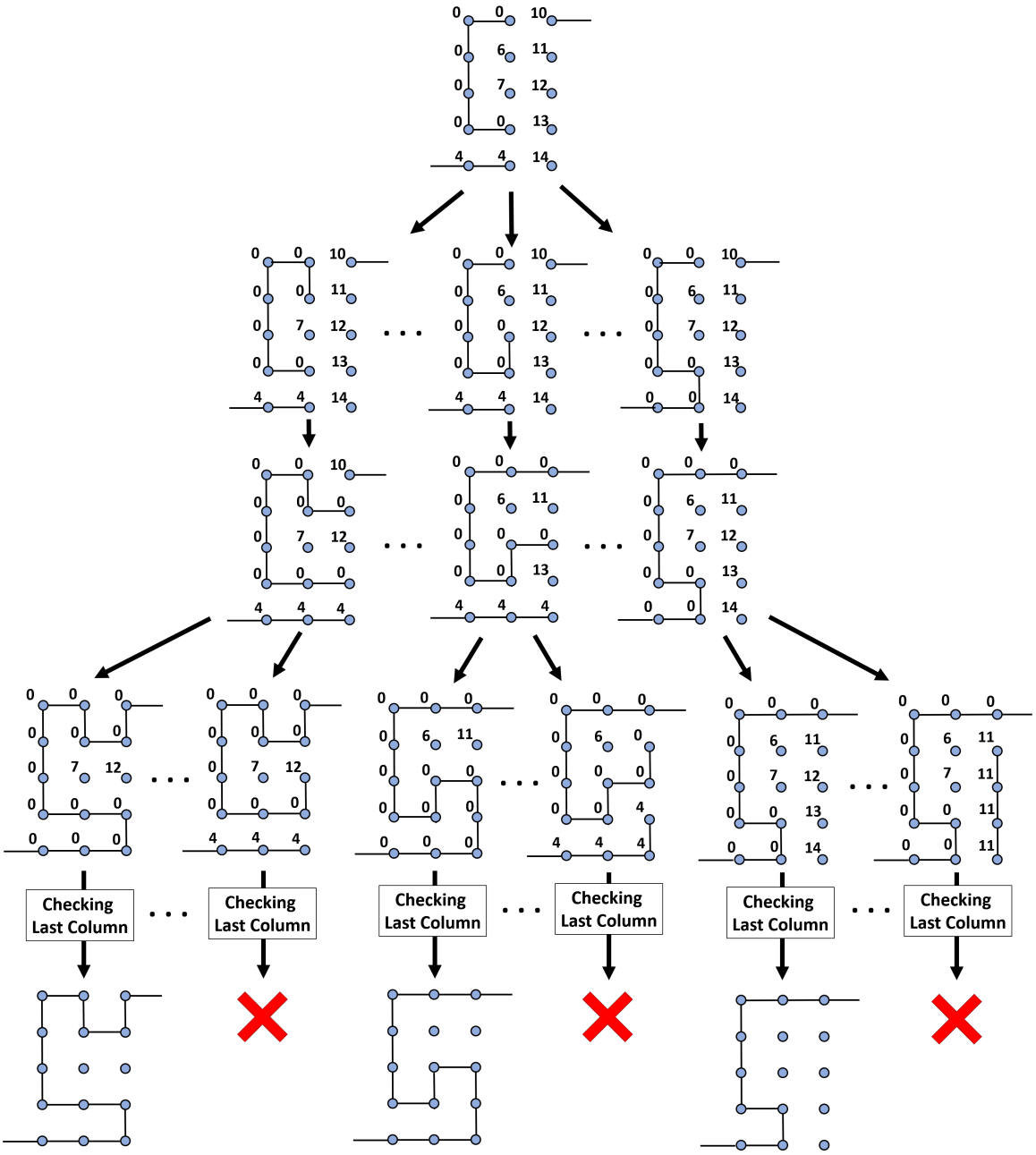


Figure 15.9: Illustration of path enumeration process in a 3x5 grid. In this figure, we show the enumeration after we generate edges on the first column. Small numbers denote SetIDs. As explained previously, top-right and bottom-left nodes are assumed as end points.

Constraint 2. $E_{i-1} \geq P \ - \ ((M-i+1)*N-1)$

In the above constraints, P denotes the desired path length, which is the desired number of edges of a path, and M and N denote the number of columns and the number of rows of a grid, respectively. The notation i denotes the column number where $1 \leq i \leq M$. Also, E_{i-1} denotes the number of edges generated between col_1 and col_{i-1}. For the boundary condition, when i=1, we set up $E_{i-1} = 0$.

The constraints can accelerate the enumeration by pruning the numerous trees. In Constraint 1, $M-i$ denotes the smallest number of edges of a connected graph we can have between column i and column M. It will be a straight path consisting of only horizontal edges between col_i and col_M. Since $P \ - \ (M-i)$ denotes the maximum value that E_{i-1} can have, if we have $E_{i-1} > P \ - \ (M-i)$, we will not get a path with the desired length P using the current E_{i-1}.

In Constraint 2, $(M-i+1)*N-1$ denotes the largest number of edges of a connected graph, which is the number of edges of a spanning tree, we can have between col_i and col_M. Since $P \ - \ ((M-i+1)*N-1)$ denotes the minimum value that E_{i-1} can have, if we have $E_{i-1} < P \ - \ ((M-i+1)*N-1)$, there are not enough edges to get the desired length P with the current E_{i-1}.

By using these constraints for path length, we can avoid having invalid edges between col_i and col_{M-1} in advance, which do not give us the desired path length, and reduce the searching time.

Remark. Using Manhattan distance, we can have a tighter constraint for Constraint 1. When we are at col_i, we calculate Manhanttan distances between nodes that have $\#NE = 1$ on this col_i and end point, which is between col_i and col_M. Among those

calculated Manhattan distances, we use the smallest distance in Constraint 1 instead of using $(M - i)$.

Chapter 16: Specifying Additional Features

When we design a maze, beside of setting up desired metric values, there can be cases that we want to specify some features directly over a maze. For example, we may want to specify the position of some maze cells directly over a grid. Also, in a hierarchical maze, we may want to choose boundaries to knock down by ourselves not by a random hierarchical connection tree. In this chapter, we explain what kind of hard constraints we can specify over a maze and how we generate mazes satisfying the hard constraints.

16.1 Hard Constraints

In this section, we introduce some hard constraints that we can specify on a maze, a solution path, and a hierarchical connection tree.

16.1.1 Maze

- **Path Constraint**: We can specify path constraints directly over a grid as shown in Figure 16.1 so that we can force a resulting maze to pass through these constraints.
- **Non-Path Constraint**: We can specify non-path constraints directly over a grid so that we force a resulting maze to have walls on corresponding places as described in Figure 16.2.

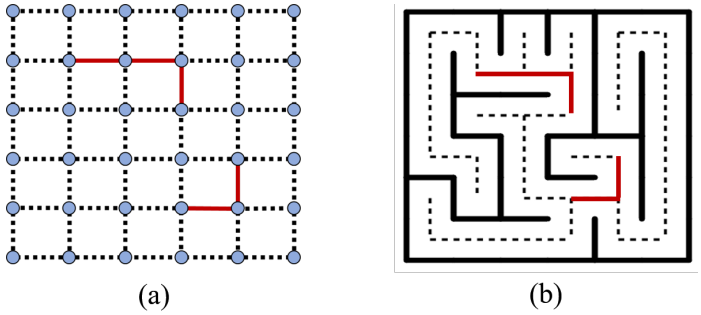


Figure 16.1: (a) Path constraints of a maze specified in a 6x6 grid (denoted by red lines). (b) Maze that has path constraints of (a).

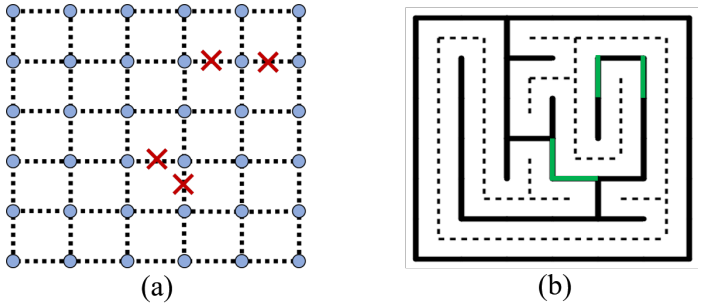


Figure 16.2: (a) Non-path constraints of a maze specified in a 6x6 grid (denoted by red X marks). (b) Maze that has non-path constraints of (a).

- **Position Constraint of Maze Cell**: We can specify position of specific maze cell on a maze. For example, we can place Cross-Junction cell around the middle of a maze as shown in Figure 16.3(a). Also, we can place four Turn cells on four corners of a maze as shown in Figure 16.3(b).

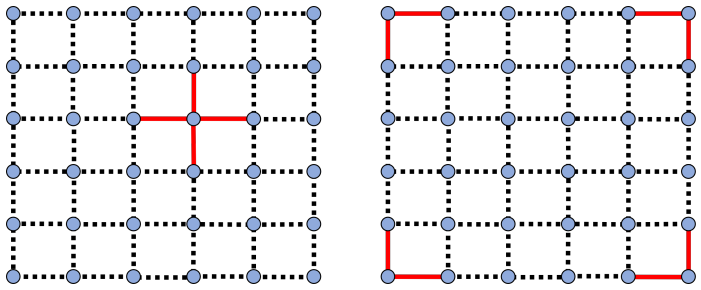


Figure 16.3: (a) Figure showing that one Cross-junction cell is specified around a middle of a 6x6 grid. (b) Figure showing turn cells that are specified on each corner of a 6x6 grid.

16.1.2 Solution Path

- **Starting Point and Ending Point**: We can specify the starting point and the ending point of a solution path so that we have resulting path between the two end points as shown in Figure 16.4.

- **Path Constraint**: We can specify paths over a grid that we force a solution path to pass through as represented in Figure 16.5.

- **Non-Path Constraint**: We can specify non-path constraint so that we force a solution path not to pass through the constraints as described in Figure 16.6.

- **Position Constraint of Maze Cell**: We can specify position of specific maze cell on a solution path. For example, we can place Turn cells and Straight cells as we want on the path.

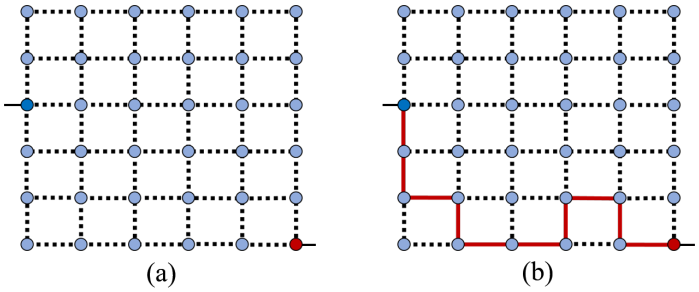


Figure 16.4: (a) The starting point (blue circle) and ending point (red circle) specified in a 6x6 grid. (b) The solution path that has two end points of (a).

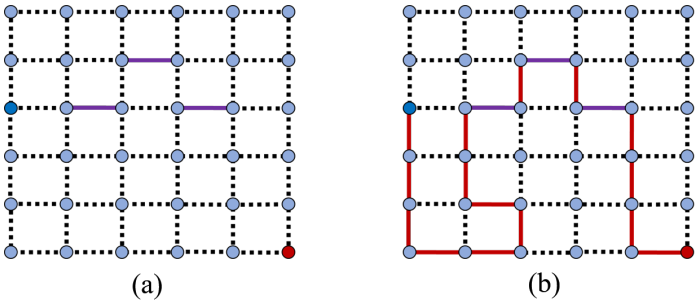


Figure 16.5: (a) Path constraints of a solution path specified in a 6x6 grid (denoted by purple lines). (b) The solution path that has path constraints of (a).

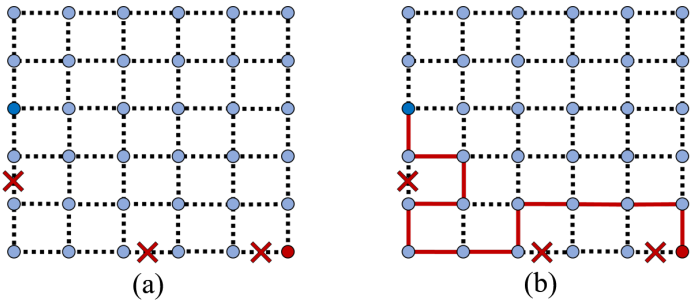


Figure 16.6: (a) Non-path constraints of a solution path specified in a 6x6 grid (denoted by red X marks). (b) The solution path that has non-path constraints of (a).

16.1.3 Hierarchical Connection Tree

- **Path Constraint**: When we create a hierarchical connection tree for a hierarchical maze, instead of randomly generating the tree, we can directly specify some paths (edges) of the tree.

- **Non-Path Constraint**: We can specify non-path constraints over a hierarchical connection tree so that the tree does not include edges corresponding to the non-path constraints.

To generate mazes satisfying the hard constraints, we can iteratively generate mazes and check whether the mazes contain the constraints. Since this brute-force method does not guarantee to have mazes with the constraints, it will take very long time to obtain the desired mazes. In the next sections, we explain how we create a maze, a solution path, and a hierarchical connection tree containing specified hard constraints.

16.2 Maze Generation with Hard Constraints

In this section, we explain how we generate a maze satisfying the constraints.

16.2.1 Inserting Hard Constraints

In our research, since a maze is represented in Edge Bit Vector (EBV) as explained in Chapter 6, we can insert constraints on a maze by fixing bits in an EBV. Now, we explain for each maze hard constraint how we fix bits in EBV.

- **Path Constraint**: When we specify path constraints on a maze, we fix bits of the corresponding paths in an EBV as 1 bits as shown in Figure 16.7.

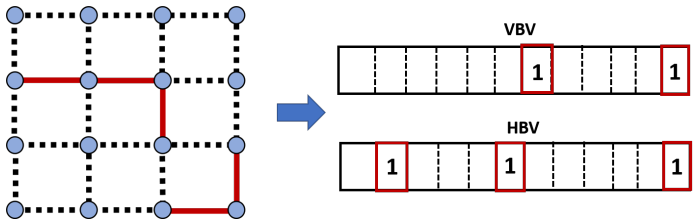


Figure 16.7: Figure showing that bits of vertical bit vector (VBV) and horizontal bit vector (HBV) corresponding to path constraints in a 4x4 grid are fixed as 1 bits.

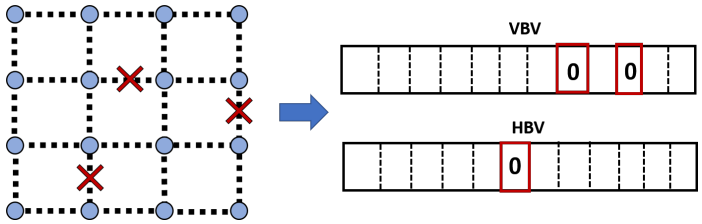


Figure 16.8: Figure showing that bits of vertical bit vector (VBV) and horizontal bit vector (HBV) corresponding to non-path constraints in a 4x4 grid are fixed as 0 bits.

- **Non-Path Constraint**: When we specify non-path constraints on a maze, we fix bits corresponding to the non-path constraints in an EBV as 0 bits as shown in Figure 16.8.

- **Position Constraint of Maze Cell**: Based on maze cell type, we fix bits in an EBV differently. For example, when we want to place one Cross-Junction cell on one node, all bits of adjacent edges of the node are fixed as 1 bits in an EBV. Likewise, when we want to place one vertically aligned Straight cell on one node, bits of top and bottom edges of the node are fixed as 1 bits, and bits of left and right edges of the node are fixed as 0 bits.

16.2.2 Constraints-based Maze Generation

When we generate a maze with desired metrics, we use a representative vector RV with the Maximum Spanning Tree algorithm (MST). After we insert hard constraints on a maze by fixing bits in an EBV, the fixed 1 bits are considered as already selected spanning tree edges on a grid, and MST selects rest of edges of the grid based on edge weights of RV. Also, during selecting edges on the grid, we force MST not to choose edges fixed as 0 bits in an EBV. After we generate a spanning tree with fixed bits of an EBV, we have a maze satisfying given hard constraints.

16.3 Solution Path Generation with Hard Constraints

In this section, we explain how we generate a solution path satisfying the constraints.

16.3.1 Inserting Hard Constraints

Here, we explain how we insert constraints in our solution path generation method. Assume that our solution path is represented in EBV like a maze.

- **Starting Point and Ending Point**: Since the starting point and the ending point must have only one adjacent edge on a path, when we have specific nodes desired for the starting point and the ending point on a grid, we set up #NE=1 for the nodes.
- **Path Constraint**: When we specify path constraints of a solution path on a grid, bits of the paths is fixed as 1 bit in an EBV.
- **Non-Path Constraint**: When we specify non-path constraints of a solution path on a grid where the path does not pass through, we fix bits of the constraints as 0 bits in an EBV.

- **Position Constraint of Maze Cell**: Based on path structures of maze cells, we specify different setting. For each path of each maze cell, we fix the corresponding bit as 1 bit in an EBV. For each non-path of each maze cell, we fix the corresponding bit as 0 bit in an EBV.

16.3.2 Constraints-based Solution Path Generation

In our solution path generation, vertical edges and horizontal edges are generated based on each column of a grid as explained in Chapter 15. On each column, for path constraints and non-path constraints, we generate edges that containing fixed 1 bits and 0 bits of an EBV. Also, for the starting point and ending point constraint, our rules that set up for the path generation in Chapter 15 will make the nodes that have #NE=1 initially get only one adjacent edge during the path generation. Then that nodes will be the starting point and ending point after a path is generated. After our path generation ends with the constraints, we have a path satisfying given hard constraints.

16.4 Hierarchical Connection Tree Generation with Hard Constraints

In this section, we explain how we insert constraints on a hierarchical connection tree and how we generate a hierarchical connection tree satisfying constraints.

16.4.1 Inserting Hard Constraints

A hierarchical connection tree also can be represented in EBV, and we can insert hard constraints by fixing bits as we did for a maze and a solution path.

- **Path Constraint**: Specified path constraints of a hierarchical connection tree have 1 bits in the corresponding slot of an EBV.

- **Non-Path Constraint**: Specified non-path constraints of a hierarchical connection tree have 0 bits in the corresponding slot of an EBV.

16.4.2 Constraints-based Hierarchical Connection Tree Generation

we can use randomized Kruskal's algorithm to generate a hierarchical connection tree on a grid. Fixed 1 bits in an EBV are considered as already selected spanning tree edges, and the algorithm will generate a spanning tree from the edges. During the edge selection in the algorithm, we force it not to choose edges specified as 0 bits in an EBV. By using this process, we have a hierarchical connection tree satisfying given hard constraints.

Chapter 17: Design-Centric Maze Generation

Previously, we've talked about how we generate desired maze topology and desired solution path. In this chapter, we introduce our design-centric maze generation method where all techniques we've developed are assembled. In this design-centric method, we allow users to input their own desired topological properties. We have detailed control over a maze 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 demonstrate the usefulness of our method, we provide several use-cases for designing computer game levels.

17.1 Motivation

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

Example 1: Consider designing a level with a search quest. As described in [4], the search quest is one where a player is asked to search a game level to collect required items. We design a game level such that the player needs to spend lots of time finding paths to each question 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 the player needs to decide which path to take next.

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 increasing running speed. 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 number 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 have the desired mazes using the current maze generation tool, we need to generate random mazes repeatedly until we get the desired ones. It is time-consuming to find such desired mazes. Therefore, in this chapter, we introduce our design-centric method, which can help users to obtain desired mazes effectively.

17.2 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.

17.2.1 Input Stage

In this stage, we input the desired properties of a maze. The desired properties are defined by properties explained in Section 4. Higher-level properties are converted to basic properties.

17.2.2 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 is given in Chapter 15.

17.2.3 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 with the desired properties and check whether the maze has the desired solution path. However, it 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 17.3, in our maze generation method, we can input some hard constraints on a grid so that a maze is forced to contain the constraints on its topology. More descriptions of this maze generation is given in Chapter 13.

17.2.4 Output Stage

In this stage, we can provide a single maze as an output. Or we can provide a set of mazes as an output so that users can choose their own best one among them. Then, the

designers can perform some post-processing over the result, such as building an actual game level on it, adding some loops, empty areas, and so on.

17.3 Applying Hard Constraints

When we define desired properties in our method, we can give hard constraints along with the desired properties as described in Chapter 16. 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 the maze topology.

17.4 Maze Design Tool

Based on our design-centric method, we have developed a maze design tool as shown in Figure 17.1. In the tool, we can specify several basic properties as input parameters, such as the size of a maze, the length of the solution path; properties of a solution path; and properties of the 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 agility vs. speed (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.

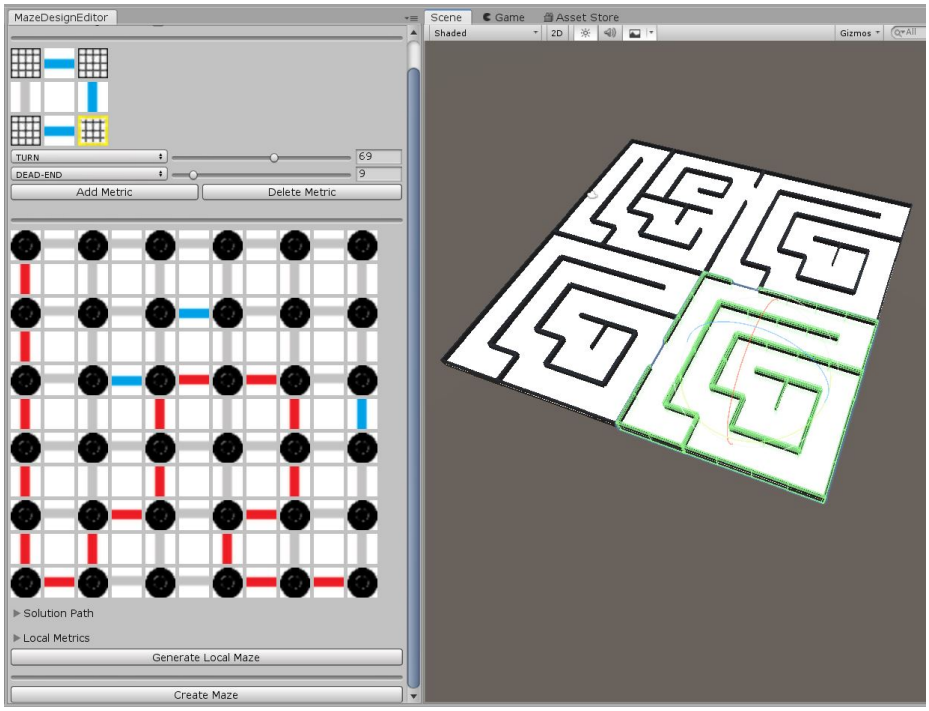


Figure 17.1: Figure showing a maze design tool we have developed.

Hard constraints also can be specified in the tool, as mentioned in Section 17.3 so that we can obtain mazes satisfying these 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 by simply clicking the 'Create Maze' button several times.

17.5 Designing Game Levels

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

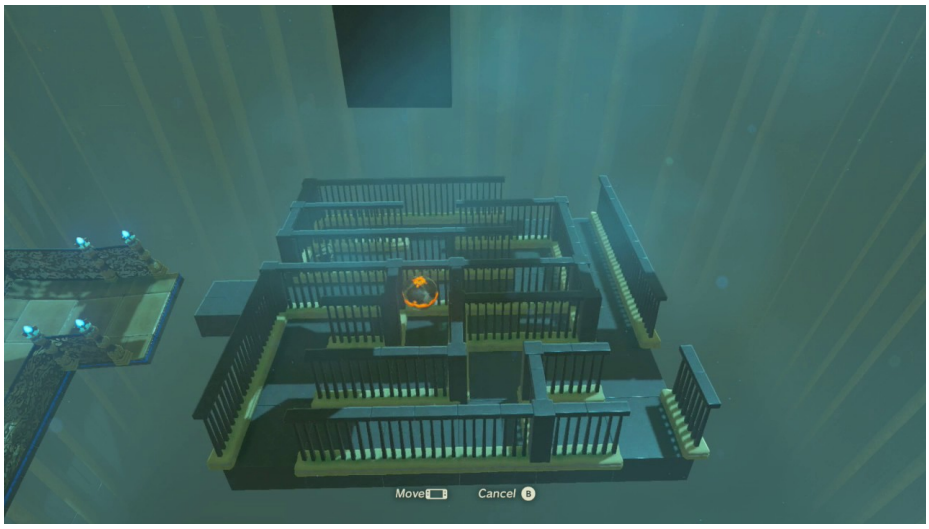


Figure 17.2: Figure showing one scene of the commercial game "The Legend of Zelda: Breath of the Wild"[27] that contains a maze-based rolling ball game.

point. This game type was also applied in the commercial game "The Legend of Zelda: Breath of the Wild"[27] as shown in Figure 17.2.

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 Chapter 4 on the platform. When we have a low AVS, the level will have long straightways 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. 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 17.3(a) shows a set of 12x12 mazes we obtained using our tool. We specified different AVS properties

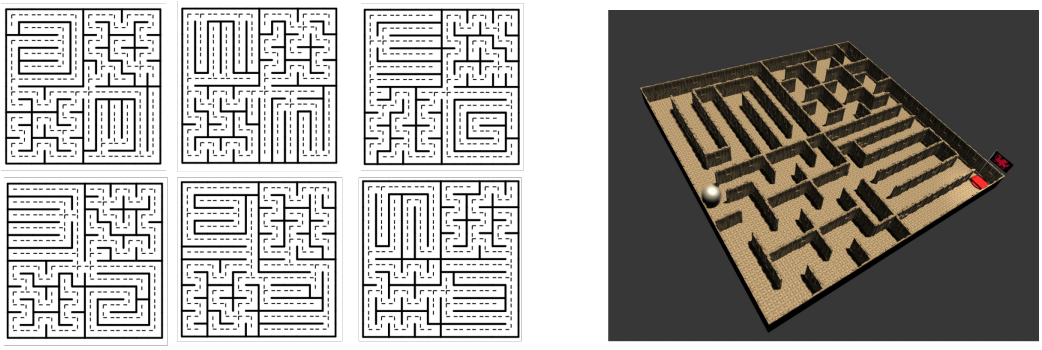


Figure 17.3: Figures showing (a) a set of mazes created by our tool for the rolling ball game and (b) an actual gameplay view. In (a), each quadrant of the maze was given different desired properties. In (b), the goal position is marked by a flag, so we need to roll the silver ball toward the flag.

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 17.3(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 17.4. 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 17.5(a), we can see a set of mazes generated

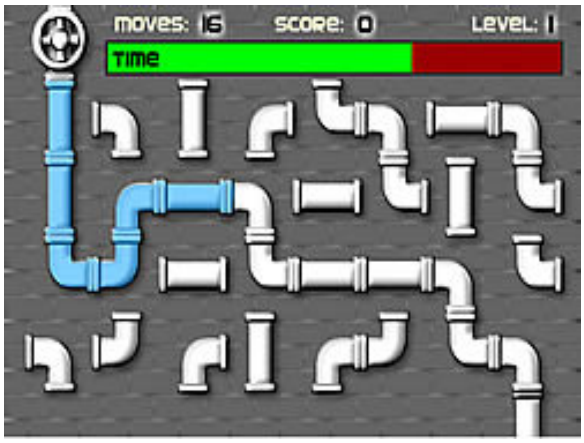


Figure 17.4: Example showing the plumbing game captured from `https://ko.y8.com/games/plumber_game`

by our tool. Figure 17.5(b) shows a pipe system where pipe parts are placed on one of the mazes in Figure 17.5(a). In Figure 13.8(c), an actual puzzle level is obtained by rotating the pipe parts. As shown in Figure 17.5(c), an unsolved puzzle level is obtained by rotating the pipe parts.

Running Game: Nowadays, as represented in [6], 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

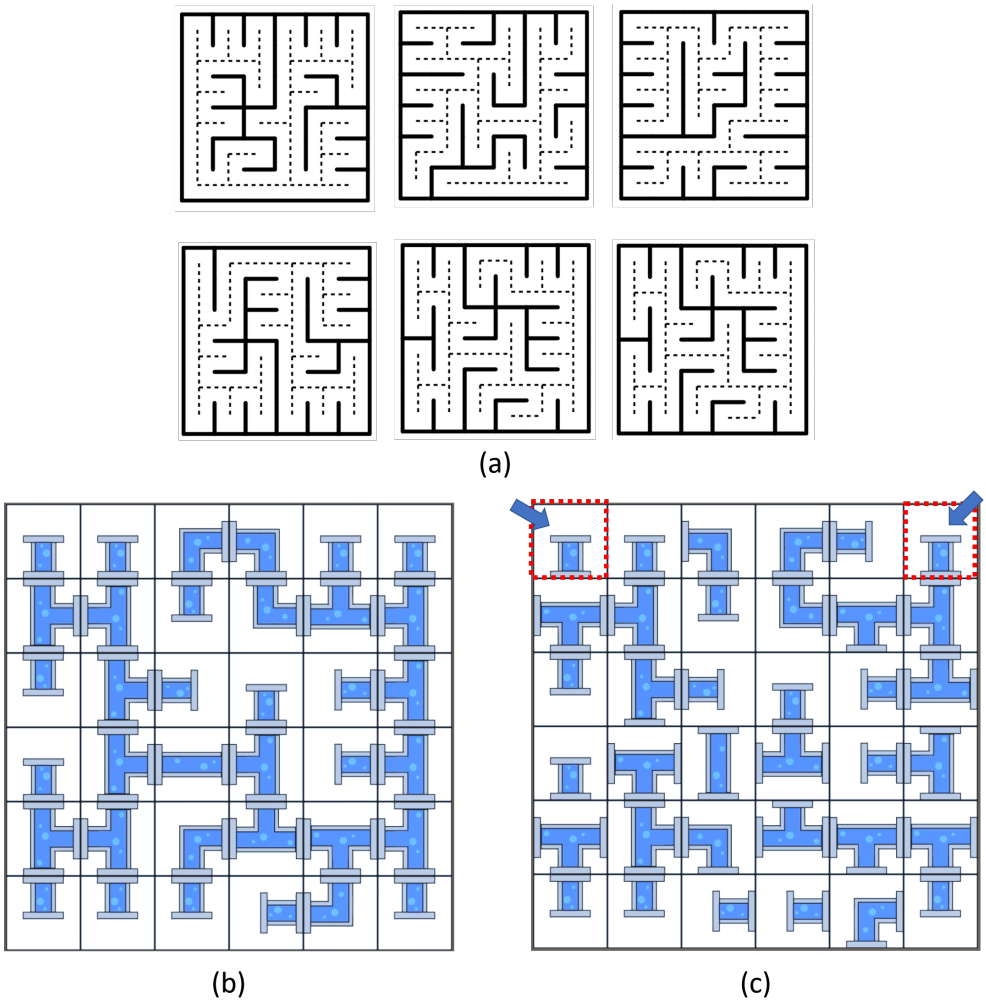
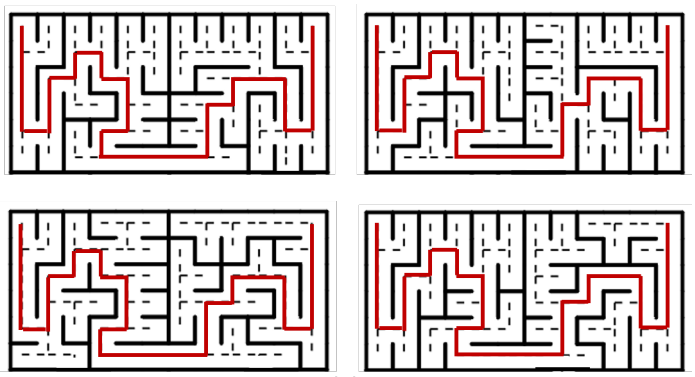


Figure 17.5: (a) A set of mazes obtained by our tool, (b) a solved plumbing game, and (c) Puzzle where we need to connect the pipe parts between the ending parts denoted by arrows.

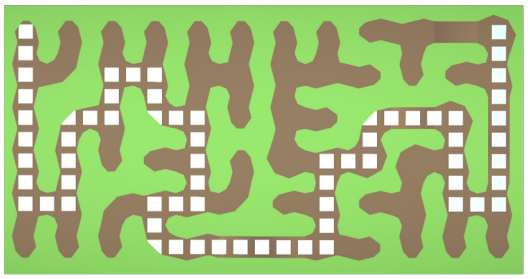
might give more freedom to a player so that the player can also explore the surrounding road map during the gameplay.

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 17.6(a) shows a set of mazes generated by our tool. In this set, we chose one maze and, as shown in Figure 17.6(b), built a level structure based on the maze. In Figure 17.6(c), we can see the actual gameplay scene with the marked running path.

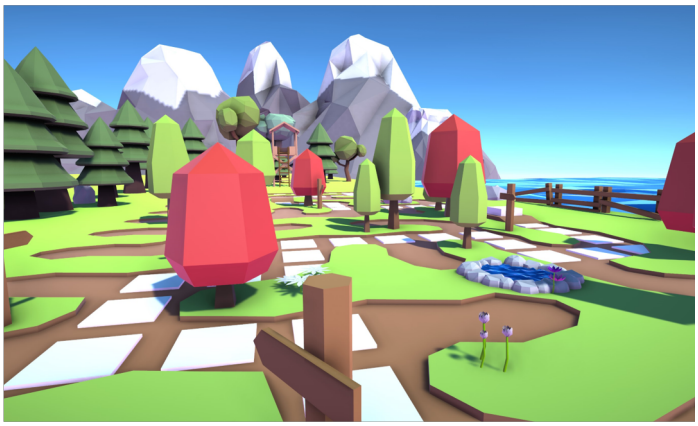
Table 17.1 provides the performance of our method in designing game levels of the above use-cases. In Table 17.1, we can see that our method needs only a few input parameters to build the desired levels. Table 17.1 also shows how accurately our method generates desired mazes. In Section 17.4, we mentioned 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 17.1, 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 17.1, 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



(a)



(b)



(c)

Figure 17.6: (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).

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

designing maze-based game levels. Then, since we can make various maze structures with little difficulty, we allow designers to find the kinds of features they're looking for.

Chapter 18: Conclusion

In this thesis, I introduced intelligent maze generation, where users can input their own topological constraints and obtain desired mazes. First of all, we investigated whether we can use existing maze generation algorithms to create desired mazes. For this investigation, we enumerated all possible spanning trees and compare them to the sampling space of the maze generation algorithms. We could see that each maze generation algorithm has different preference over the sampled mazes. Second, we developed a method, in which it chooses the best algorithm amongst based on input desired properties and use the best one in an SBPCG approach to find desired mazes. This method always chooses the algorithm which gives us the best chance, but there were still some mazes that are very hard to find by any existing algorithm. To create any valid maze, we developed a new method which enumerates the spanning tree space. Using the representative vectors, we obtain any possible desired mazes with very good performance. In our intelligent maze generation, we also developed a path generation method where users can input desired path properties and obtain the corresponding paths. This path generation helps to set up the solution path on a maze. To demonstrate the effectiveness of our intelligent maze generation method, we provided several use-cases of building an actual game level and showed how our method could help users to design their desired content. We observed that our method helps to generate the desired content quickly with a few input parameters. Our maze generation has a promising

potential to be applied to design contents in various fields, especially the field of computer game content generation.

Bibliography

[1] *Caterpillar graph*. https://en.wikipedia.org/wiki/Caterpillar_tree.

[2] *Hilbert curve*. https://en.wikipedia.org/wiki/Hilbert_curve.

[3] *Micromouse Event*. http://micromouseusa.com/.

[4] *RPG Design Patterns*. https://rpgpatterns.soe.ucsc.edu/doku.php.

[5] *Spanning Tree*. https://en.wikipedia.org/wiki/Spanning_tree.

[6] *Zwift Run*. https://zwift.com/en/run/.

[7] C. Adams and S. Louis. Procedural maze level generation with evolutionary cellular automata. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Nov 2017.

[8] David J. Aldous. The random walk construction of uniform spanning trees and uniform labelled trees. *SIAM J. Discret. Math.*, 3(4):450–465, November 1990.

[9] D. Ashlock. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 289–296, Aug 2010.

[10] D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):260–273, September 2011.

[11] Christopher Berg. *Amazeing Art*. http://amazeingart.com.

[12] Robert Bosch, Sarah Fries, Mäneka Puligandl, and Karen Ressler. From path-segment tiles to loops and labyrinths. In *Proceedings of Bridges 2013: Mathematics, Music, Art, Architecture, Culture*, July 2013.

[13] A. Broder. Generating random spanning trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, pages 442–447, Washington, DC, USA, 1989. IEEE Computer Society.

[14] Jamis Buck. *HTML 5 Presentation with Demos of Maze Generation Algorithms.* `www.jamisbuck.org/presentations/rubyconf2011/index.html`.

[15] Jamis Buck. *Mazes for Programmers: Code Your Own Twisty Little Passages.* Pragmatic Bookshelf, 2015.

[16] S. Chaiken and DJ. Kleitman. Matrix tree theorems. 1978.

[17] Wen-Shou Chou. Rectangular maze construction by combining algorithms and designed graph patterns. *GSTF Journal on Computing (JOC)*, August 2016.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

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

[20] Peter. Gabrovšek. Analysis of maze generating algorithms. *IPSI Transactions on Internet Research.*, 15(1), January 2019.

[21] Craig S. Kaplan. The design of a reconfigurable maze. In *Proceedings of Bridges 2014: Mathematics, Music, Art, Architecture, Culture*, August 2014.

[22] P. H. Kim and R. Crawfis. Intelligent maze generation based on topological constraints. In *2018 7th International Congress on Advanced Applied Informatics*, July 2018.

[23] Paul Hyunjin Kim, Jacob Grove, Skylar Wurster, and Roger Crawfis. Desing centric maze generation. In *The 10th Workshop on Procedural Content Generation*, August 2019.

[24] A. Kozlova, J. A. Brown, and E. Reading. Examination of representational expression in maze generation algorithms. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 532–533, Aug 2015.

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

[26] D. Maung and R. Crawfis. Applying formal picture languages to procedural content generation. In *2015 Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES)*, pages 58–64, July 2015.

[27] Nintendo. *The Legend of Zelda: Breath of the Wild*, 2017.

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

[29] C. C. Palmer and A. Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 379–384 vol.1, June 1994.

[30] Hans Pedersen and Karan Singh. Organic labyrinths and mazes. In *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, NPAR '06, pages 79–86, New York, NY, USA, 2006. ACM.

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

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

[33] G. R. Raidl and B. A. Julstrom. Edge sets: an effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, 7(3):225–239, June 2003.

[34] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

[35] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 4:1–4:7, New York, NY, USA, 2010. ACM.

[36] L. Wan, X. Liu, T. Wong, and C. Leung. Evolving mazes from images. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):287–297, March 2010.

[37] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 296–303, New York, NY, USA, 1996. ACM.

[38] Jie Xu and Craig S. Kaplan. Vertex maze construction. *Journal of Mathematics and the Arts*, November 2006.

[39] Jie Xu and Craig S. Kaplan. Image-guided maze construction. *ACM Trans. Graph.*, 26(3), July 2007.