

MAD MAZE GENERATION AND DESIGN CONSIDERATIONS

by

Johnathon Henke

Copyright by Johnathon Henke 2023

All Rights Reserved

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Computer Science).

Golden, Colorado

Date _____

Signed: _____

Johnathon Henke

Signed: _____

Dr. Dinesh Mehta
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____

Dr. Iris Bahar
Professor and Head
Department of Computer Science

ABSTRACT

We define several logic mazes and their state graph representations with the goal of generating additional instances of each type of maze. Local search generation was found to be the most effective method of maze generation; thus, defining an objective function that considers maze characteristics is a focus of this work. Modeling logic maze puzzles as state graphs enables one to use the same objective function to compare and generate instances of multiple different types of logic mazes. We specify assorted features that one may desire in a maze, provide implementation details, and give example objective functions that can be used to generate mazes with desired characteristics.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vi
LIST OF TABLES	viii
CHAPTER 1 INTRODUCTION	1
1.1 Mazes	1
1.2 “Mad” Mazes	1
1.3 Preview	2
CHAPTER 2 BACKGROUND	4
2.1 Mad Mazes Project	4
2.2 Modeling	5
2.2.1 Classic Maze State Graphs.....	5
2.2.2 Jumping Mazes and Variants	7
2.2.3 Arrow Mazes and Variants	10
2.2.4 Connections Mazes	14
2.2.5 Multiplayer Mazes and Variants	16
2.2.6 Misc. other Mad Mazes	21
CHAPTER 3 MAZE CHARACTERISTICS	22
3.1 General Maze Characteristics	22
3.1.1 Paths, Shortest Paths, and Number of Nodes	24
3.1.2 Branching	25
3.1.3 Reachability.....	25
3.1.4 Cycles, Holes, and Whirlpools	26
3.1.5 Decisions, Required Vertices, Bridges/Dominance	29
3.1.6 Long False Paths (Decisions continued).....	31
3.2 Mad Maze Specific Characteristics	32
3.2.1 Color.....	33

3.2.2	State Changes.....	33
3.2.3	Revisiting Locations and Doubling Back	34
3.3	Component Graph.....	38
CHAPTER 4	IMPLEMENTATION AND SEARCH	41
4.1	Implementation	41
4.1.1	Program Infrastructure.....	42
4.2	Fully Random Generation.....	42
4.3	Intelligent Random Generation	43
4.3.1	Automation of manual generation	43
4.3.2	Local search	45
4.3.3	Search Options.....	46
4.3.4	Neighbor States	48
4.4	Example Search Function	50
4.5	Results.....	51
4.5.1	Jumping Maze - Jumping Jim's Encore.....	52
4.5.2	Arrow Maze - Apollo's Revenge	58
4.5.3	Connections Maze - Grandpa's Transit Map	60
4.5.4	Multiplayer Maze - Spacewreck.....	64
CHAPTER 5	CONCLUSIONS AND FUTURE WORK	71
5.1	Future Work	71
5.1.1	Human Maze Solving.....	71
5.1.2	Additional Logic Mazes	72
5.1.3	State Space Puzzles	72
5.1.4	Explore other optimization techniques	73
5.1.5	State Graph to Maze Instance.....	73
5.1.6	Applications in pedagogy: Puzzle-based Learning	73
5.2	Conclusions.....	73
REFERENCES	75

LIST OF FIGURES

Figure 1.1	Maze #1 of <i>Mad Mazes</i> (Abbott, 1990, 5).....	2
Figure 2.1	A classic maze and one possible graph representation	6
Figure 2.2	A basic jumping maze.	7
Figure 2.3	A jumping maze with red, circled numbers that change the direction of movement.	8
Figure 2.4	Graph model of the jumping maze instance shown in Fig. 2.3.....	11
Figure 2.5	A simple arrow maze.	12
Figure 2.6	The left shows a colored arrow maze. Solvers must alternate colors when moving through this maze. The right instance depicts a colored arrow maze with diagonal state changes.	12
Figure 2.7	Arrow Hockey instance from SuperMazes Abbott (1997).	14
Figure 2.8	Connections maze with two colors and four line types.	15
Figure 2.9	Graph model of Fig. 2.8	17
Figure 2.10	An example Spacewreck maze.	18
Figure 2.11	State graph model of the maze in Fig. 2.10	20
Figure 3.1	A jumping maze generated by a function that maximizes state changes without a limit on the number of circled squares.....	34
Figure 3.2	The Spacewreck instance given in Mad Mazes (Abbott, 1990, 32).....	35
Figure 3.3	The top section of Abbott's Grandpa's Transit Map connections maze (Abbott, 1990, 19). The arrow points to the village mentioned when discussing transit lines exited from the start.	36
Figure 3.4	Jumping maze with an abundant use of doubling back.	37
Figure 3.5	Component graphs of the connections and spacewreck mazes given in Figures 2.8, 2.9, 2.10, 2.11.	38

Figure 3.6	Component graph of Fig. 3.4 with single-vertex unreachable or unreachng sccs removed that did not have both incoming and outgoing edges.....	40
Figure 4.1	Potential inheritance scheme	41
Figure 4.2	Exceptional jumping maze instance.	56
Figure 4.3	Component graph of the excellent jumping maze instance.	57
Figure 4.4	Best Apollo's Revenge maze instance.	60
Figure 4.5	Component graph of the best Apollo's Revenge instance. Compared to the jumping maze instance in Fig. 4.3, it has many more dead ends and smaller holes.	61
Figure 4.6	Granpa's Transit Map instance one. Note that E is an isolated village without any transit lines. This means we could potentially increase the number of transit lines when generating these instances. One of the two shortest paths is A F L Q M L R N S R W X S O T S Y d c X d Y S T O S X W R S N R L M Q W b f.	64
Figure 4.7	Maze one of the connections mazes - exceptional holes and a whirlpool combined with a large number of required decisions.	68
Figure 4.8	The best Spacewreck instance (three color).....	69
Figure 4.9	The component graph associated with the best Spacewreck instance given in Fig. 4.8.	70

IST OF TABLES

Table 4.1 Jumping generation maze results are shown below with the primary components of the scores highlighted in bold . SP = shortest path, BH = black hole, WH = white hole, R or R = Reachable or Reaching, R and R = Reachable and Reaching. The hole entrances are only to the largest hole. The required decisions is the sum of the required vertices out degrees, which represents the minimum quantity of choices maze solvers will consider. Fwd and bwd decisions are the forward and backward decisions, and the other metrics are previously specified in this chapter and Chapter Two.	53
Table 4.2 Apollo's Revenge generation maze results are shown below with the primary components of the scores highlighted in bold . SP = shortest path, BH = black hole, WH = white hole, R or R = Reachable or Reaching, R and R = Reachable and Reaching.....	59
Table 4.3 Grandpa's Transit generation maze results are shown below. There are 128 vertices in each state graph, and Two SPs Addback is the $10 * n$ given from having two shortest paths. When a line/village is visited multiple times, this is counted in Village/Line Doubling. The color traits penalize the score for unequal color distributions.	62
Table 4.4 Spacewreck generation maze results are shown below. Num Colors is the number of colors used in creation of the maze instance (two or three). Room repeating is the count of the same room being visited multiple times, Same Room is the count of the number of times the two players must enter the same room, and the colors are penalties for uneven color distributions.....	66

CHAPTER 1

INTRODUCTION

1.1 Mazes

Mazes have long fascinated humanity. In one of the world's first civilizations in Sumer, we have discovered mazes and labyrinths. Millennia ago, isolated civilizations resulted in unique alphabets, languages, and dialects. Yet mazes and labyrinth designs have existed for thousands of years across many civilizations: from rock carvings in Sardinia (2500-2000 BC) to Padugula in South India (circa 1000 BC) to Val Camonica in Italy (750-550 BC), (Fisher and Gerster, 2000, 12). Mazes and labyrinths have inspired stories and legends, such as Daedalus, Theseus, and the legendary labyrinth in Greek mythology. The prevalence of mazes across history speaks to their extraordinary presence. "Once seen, a maze cannot be ignored. It draws you into it like a magnet, then proceeds to puzzle, infuriate, and delight in turn until its goal is reached. Mazes have been exerting this maddening fascination for thousands of years, and evidence of them is to be found in different civilizations all over the world (Fisher and Gerster, 2000, 12)."

This work involves a more recent development in the history of the maze: the *Mad Maze*, or logic maze.

1.2 “Mad” Mazes

The term “Mad” maze was coined by Abbott in his 1990 book of the same name. A more widespread and common term is logic mazes Abbott (1999). These mazes introduce additional rules to basic mazes such that they become more of logic puzzles rather than classic path-finding mazes. Another appropriate name for them is “Multi-state” mazes because it is possible to revisit the same location in the maze multiple times in different “states” (in many logic mazes, it is required to do so).

Consider the first maze in Abbott's Mad Mazes. It is a basic classic maze with one additional rule: after passing through a red location, one must pass through a blue location and alternate colors until reaching the exit.

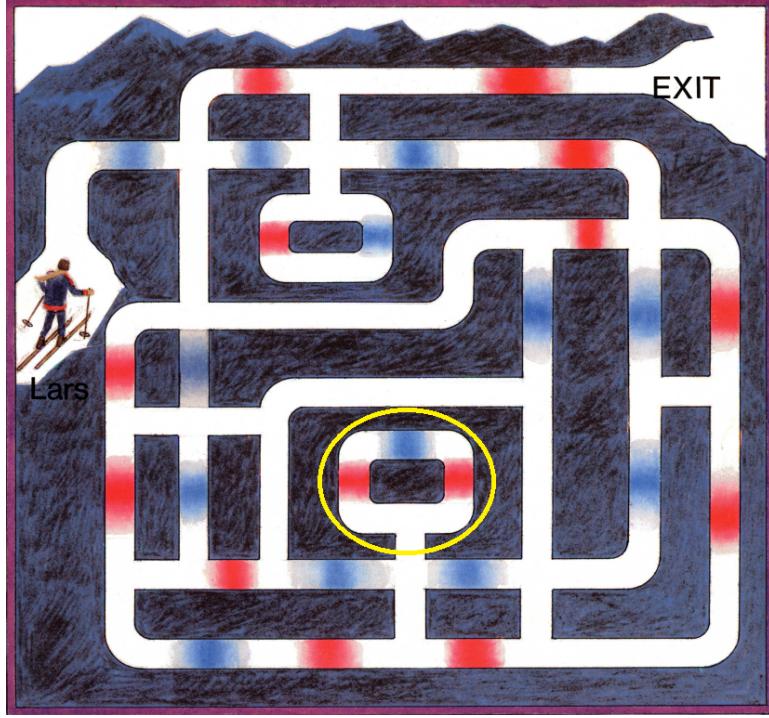


Figure 1.1: Maze #1 of *Mad Mazes* (Abbott, 1990, 5).

The defining feature of this maze is the lower loop that alternates red blue red highlighted above. Although this loops appears to be a superfluous feature, it is actually required to pass through this lower loop in order to reach the finish. Abbott mentions this when he designed this maze (Abbott, 1990, 38). Adding these simple rules to classic mazes changes them from a simple path-finding puzzle into a logic puzzle and profoundly increases their interest. Logic mazes such as these can be directly modeled as graphs to be solved, which leads to the purpose and motivation of this work

1.3 Preview

In Chapter Two, we provide the background and motivation for this work, along with the state graph models of some logic mazes. Chapter Three introduces several maze characteristics that can be used in an objective function to provide a rating for state graphs and the high-level instances of each logic maze. Implementation details and an example objective function to generate mazes with specific characteristics are introduced in Chapter Four, along with search strategies (simulated annealing, local search, random restart, etc.).

In Chapter Five, we summarize the paper and conclude with potential future work.

CHAPTER 2

BACKGROUND

2.1 Mad Mazes Project

Graph theory is a fundamental component of the undergraduate Algorithms class at Colorado School of Mines, and one of the most important concepts covered in the class involves the concept of “graph modeling”, or taking an unknown problem that one does not know how to solve, modeling it as a graph, and running a well-known graph algorithm (ideally without modifications) to solve the problem. An example of this would be the following problem:

Consider a set of movies M_1, M_2, \dots, M_k and a set of customers, each of whom indicates the two movies they would like to see this weekend. Movies are shown on Saturday evening and Sunday evening. Multiple movies may be screened at the same time. You must decide which movies should be televised on Saturday and which on Sunday, so that every customer gets to see the two movies they desire. Is there a schedule where each movie is shown at most once? Design an efficient algorithm to find such a schedule if one exists (Skiena, 2008, 188).

This problem can be solved by modeling it as a graph. Model movies as vertices and preferences as edges. If a person wants to watch movies M_1 and M_2 , let there be an edge (undirected, unweighted) between M_1 and M_2 . After creating this graph, attempt a 2-coloring using a BFS.

Graph modeling is a crucial concept, which lead to the Maze project. While teaching this class, Dr. Mehta encountered Robert Abbott’s book, *Mad Mazes*. This work (which appears to have been designed as a puzzle book) contains 20 Mad Mazes, and Dr. Mehta realized that many of these puzzles serve to directly teach this concept of graph modeling, and thus the Mad Mazes project was born. Students were given a maze from the book with all of its additional rules and complexity beyond a classic maze, asked to model it as a graph and run a well-known graph algorithm completely unmodified (students are

encouraged to use a graph library to this end) to solve the maze, which equates to finding the shortest path from the start to the finish. Some complications arose; however, from this. Abbott provides only one instance of each maze type in the Mad Mazes book, and solving Abbott’s instance by hand proves to be a grueling task. From a pedagogical perspective, it would be better to have smaller instances of each maze that students can use to familiarize themselves with the rules and additional instances that can be used to test each student’s implementation of their algorithm (or model), which was the direct motivation for this work: how can one generate logic mazes of a desired size and difficulty?

2.2 Modeling

As mentioned in the previous section, all mazes (classic and logic) can be represented as a state graph where each vertex in the graph represents a state and edges denote possible movements between states. For classic mazes, one’s location is the only state that needs to be tracked, and there are several methods to track the state to form vertices. One method involves modeling each “decision point” as a vertex (typically intersections) and corridors between the decisions points as edges connecting the vertices (see the figures below). Note that all classic mazes are undirected—it is always possible to retrace one’s steps. This is often not the case in logic mazes.

It is important to note the graph representation abstracts away the visual structure of the maze. A maze’s shape can increase its difficulty and additionally confuse its solvers, especially with life-size mazes such as those constructed in theme parks and other places. However, in this work, we focus on the structural elements of the maze encoded into its state graph. The physical shape of the maze and its effect on the human psyche is out of the scope of this work.

2.2.1 Classic Maze State Graphs

As stated previously, any classic maze can be solved by modeling it as a graph and finding the shortest path through the maze using a BFS. The shortest path through the example maze given in Fig. 2.1 is S, 1, 3, 5, 8, 10, 12, F. Note this is the only path that

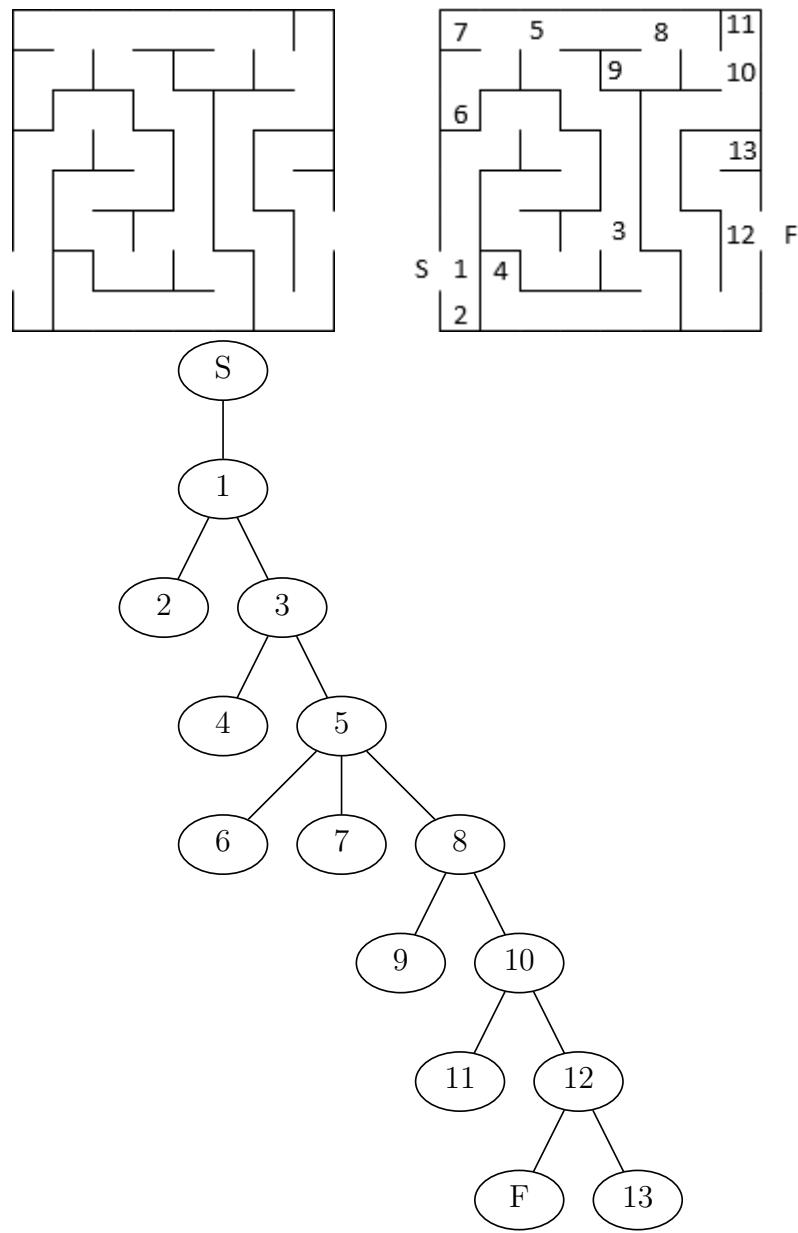


Figure 2.1: A classic maze and one possible graph representation

does not repeat any vertices, thus it constitutes a *simple* path.

2.2.2 Jumping Mazes and Variants

One of the more common types of logic mazes is a jumping maze. The first jumping maze Abbott introduces is maze #7 Jumping Jim (Abbott, 1990, 14-15)—this simple version of a jumping maze consists of a simple numeric grid. The number on each cell indicates how far one must move (horizontally or vertically, but not diagonally) from that cell. Starting from the upper left hand corner, the challenge is to find the minimum number of “jumps” to reach the bottom right hand corner, which is the finish (note it is the number of jumps that we try to minimize and not the total length of the jumps).

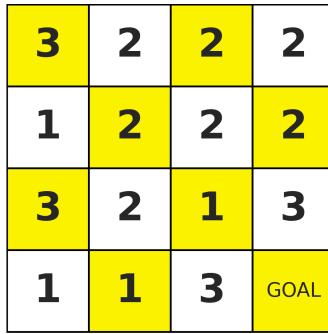


Figure 2.2: A basic jumping maze.

The classic jumping maze has a simple model. Create an unweighted, directed graph G . Add vertices representing each cell on the grid and directed edges from each vertex to the cells it is possible to move to. For example, the vertex for (1, 1) in the example shown in Figure 2.2 would have directed edges to (4, 1) and (1, 4). After completing the graph model, a BFS will find the shortest path from the start to the finish because it is an unweighted graph (Dijkstra’s or another weighted graph algorithm is not necessary).

Due to this puzzle’s simple state graph reduction, classic jumping mazes were not used for the graph modeling project. Instead, the jumping maze variant introduced in maze #15 Jumping Jim’s Encore was used (Abbott, 1990, 26-27). This variant introduces circled numbers that change the direction of movement. If one lands on a circled number while

moving horizontally or vertically, then one's direction of movement changes to diagonal until reaching another circled number, at which point it reverts to vertical/horizontal. This small change introduces a concept of "movement state" into the maze, making it possible to visit the same cell multiple times, once in the state of vertical/horizontal movement, (which we refer to as cardinal movement) and again in the state of "diagonal" movement.

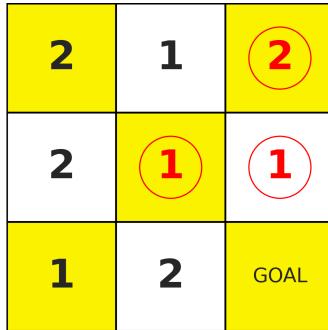


Figure 2.3: A jumping maze with red, circled numbers that change the direction of movement.

In order to handle this new complication, we must either modify the model or the algorithm. Modifying the algorithm would involve adding conditional statements to the BFS to allow vertices to be visited twice, once cardinally and once diagonally. However, we prefer to modify the graph instead of the algorithm, for several reasons. This concept of "modeling" a logic maze can also be thought of as a reduction to a state graph (or a maze without state). This allows us to use similar components in the maze design function and consider similar traits when rating the difficulty of multiple logic mazes, which is discussed in detail in chapters 3 and 4. There is also a notion of "reinventing the wheel" when creating a new algorithm to solve the problem. Graph theory is an expansive subject with extensive research and existing algorithms. Encoding the complexities of a logic maze into the graph representation enables one to use the myriad of already-existing graph algorithms to not only solve the maze, but also study (and rate) its characteristics. Compared to the alternative of re-implementing every graph algorithm one would like to use with small modifications, the former option is much more attractive and efficient.

Combining the modeling idea with a graph library greatly simplifies implementation, enabling one to use all of the conveniences that come with graph libraries, such as the ability to treat the graph as an abstract object neglecting implementation details along with a bug-free implementation of many common graph algorithms. Additionally, as previously stated, we are teaching graph modeling and in fact require students to modify the graph instead of the algorithm, and thus we encourage students to use a graph library when solving these logic mazes.

Given that we are modifying the graph and not the BFS algorithm, consider the following model for the diagonal jumping maze variant M :

Let G be a directed, unweighted graph. For each cell s on the grid with uncircled number n in M , create two vertices in G , $s1$ and $s2$, with $s1$ representing the cardinal state of s and $s2$ representing the diagonal state of s . Add directed edges from $s1$ to the cardinal vertices of distance n from s considering cardinal movement. Add directed edges from $s2$ to the diagonal vertices of distance n considering diagonal movement. The process is similar for circled numbers, except the outgoing edges connect vertices in the two different movement mazes. Add directed edges from $s1$ to the diagonal vertices of distance n considering diagonal movement. Add directed edges from $s2$ to the cardinal vertices of distance n considering cardinal movement.

In the grid, there will be one cell without a number; this is the goal cell. In this variant of a jumping maze, the goal cell is always the bottom right corner (though this is flexible and may not be the case in other variants). Create only one vertex to represent the goal state (instead of two as for other cells). When it is possible to reach the goal state in one move whether one is in the cardinal state or the diagonal state, add an edge to this single goal vertex.

To remove the notion of “state”, which is the movement type, we have expanded the number of vertices in the graph relative to the number of possible states. There are two states per cell, and thus double the number of vertices in the state graph (minus one for the goal state), two for each location on the grid and one for each possible state. In effect,

we have created two *distinct* mazes, the cardinal maze and the diagonal maze, and the circled numbers serve as connections between the two. This is common for many logic mazes with state changes. often, we expand the graph as compared to the puzzle to encode the puzzle’s complexity, and the additional space consumed is a constant times the number of vertices that would be in the direct translation of the puzzle to a graph. This is not true for all logic mazes; in some cases, the expansion of nodes is a polynomial factor of the size of the original maze instead of a constant (see later sections on step-change (Alice) mazes or multi-player mazes).

After constructing the graph, a BFS will find (one of) the shortest path(s) from the start to the goal vertex (note the start is always the upper left cell in the cardinal state in this variant). See Fig. 2.4 for an example graph. The naming convention of the vertices is (state, row, column) where 1 indicates a cardinal state and 2 indicates a diagonal state. Ignore the colors and shapes for now, these will be explained further in a later chapter.

2.2.3 Arrow Mazes and Variants

Another common type of logic maze is the arrow maze. Similar to jumping mazes, the maze is constructed on a grid, with the start as the upper left hand cell and the finish as the lower right corner. Instead of numbers on the cells; however, there are arrows. There are eight possible orientations of an arrow, and the easiest method to denote each orientation is to apply the eight principal directions. An arrow that points “North” points towards the top border of the maze and the other directions can be inferred from this.

Movement through arrow mazes is as follows: one can choose to move to any one of the cells the current arrow points at. As seen in Fig. 2.5, the top left arrow points downwards, thus it is possible to move from the top left to any other cell in the first column.

Arrow mazes are modeled in a similar manner to jumping mazes. Let G be a directed, unweighted graph. For each cell s on the grid of arrow maze M , add a vertex representing this cell and directed edges from this vertex to the vertices representing the cells the arrow in s points at. Now that we have reduced the arrow maze to a state graph, we can run a

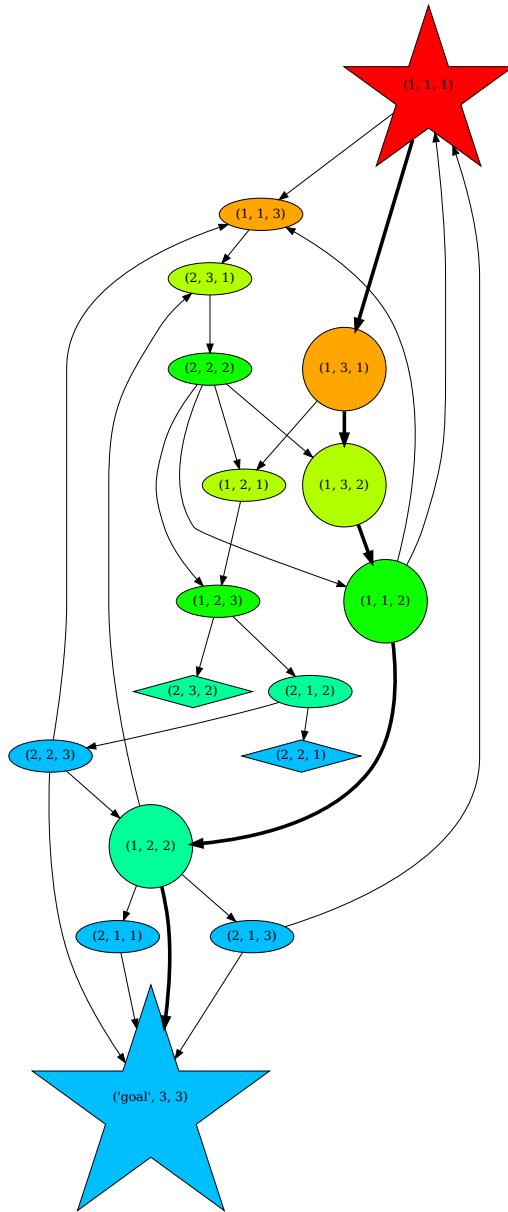


Figure 2.4: Graph model of the jumping maze instance shown in Fig. 2.3

BFS from the start to find the shortest path from the start to the goal.

Abbott introduces several variants on the basic arrow maze, the two in Mad Mazes being the Apollo and Diana and Apollo's Revenge mazes (Abbott, 1990, 16-17). Apollo and Diana introduces a coloring scheme to the basic arrow maze - the arrows are colored red and blue, and one must alternate between red and blue arrows when moving through the maze (see Fig. 2.6). Adding a coloring scheme of any kind (red-blue, red-green-blue, etc.) does not change the model significantly—when iterating over the cells pointed at by

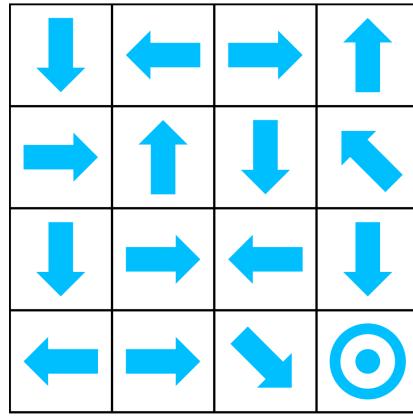


Figure 2.5: A simple arrow maze.

the current arrow, only add an edge from the current vertex to the vertex representing the child if the color is valid.

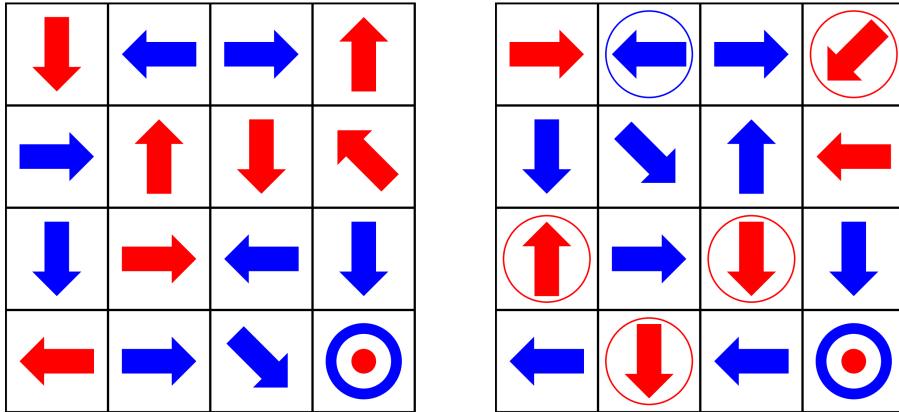


Figure 2.6: The left shows a colored arrow maze. Solvers must alternate colors when moving through this maze. The right instance depicts a colored arrow maze with diagonal state changes.

Apollo's Revenge introduces a state change similar to the circled numbers in jumping mazes (see Fig. 2.6). When moving through the maze, circled arrows change the direction of movement from forward to backward: when one lands on a circled arrow while moving normally, one moves out of the *tails* of the arrows, that is, backward, until reaching another circled arrow at which point one moves normally again. Apollo's Revenge

combines a coloring rule with a state-change in the final maze.

The modeling for a state-changing arrow maze is very similar to that of a state-changing jumping maze. Using the model given in the jumping maze state-change section, create a subgraph for each of the types of movement, and the circled arrows connect the two subgraphs to form the overall state graph.

Abbott introduces what he considers the best of his arrow mazes in SuperMazes, the sequel to Mad Mazes, and he coins this variant “Arrow Hockey.” In an Arrow Hockey maze, the rules of a basic arrow maze apply, except that there is a dime on one of the cells. The goal of the maze is to bump the dime from cell to cell until moving it onto the goal cell. Movement rules are identical to a basic arrow maze (one can move to any cell that the current cell’s arrow points at) except that one may not travel over the dime. Although one cannot travel over the dime, it is allowed to end a move on the same cell as the dime, in which case the dime is bumped one cell forward in the direction one came from when landing on the cell with the dime. For example, if the dime was at cell $(3, 3)$ and the player was at $(3, 1)$ with a rightward pointing arrow and chose to move onto the cell with the dime, $(3, 3)$, because the player came from the left the dime would be bumped to cell $(3, 4)$.

The location of the dime changes the rules of the maze. Therefore, each possible location of the dime represents a different maze. To entirely remove state and reduce this maze to a state graph, we will have n subgraphs, one for every possible location of the dime (which is assumed to be every cell in the maze). This means the graph representation of an arrow hockey instance will have n^2 vertices, where n is the number of cells in the original maze. If we consider a $n \times n$ grid version of the arrow hockey maze, it will have n^4 vertices. This reduction expands the number of vertices and edges (and thus the memory consumed) by a polynomial factor based on the size of the original maze, compared to the constant factor we have seen for previous state changing mazes. Arrow hockey mazes with even only a thousand cells can become difficult to work with due to the size of the state graph.

Mazes such as arrow hockey beg the question of whether or not it is better to modify the algorithm used to solve the maze and simulate it directly instead of performing the

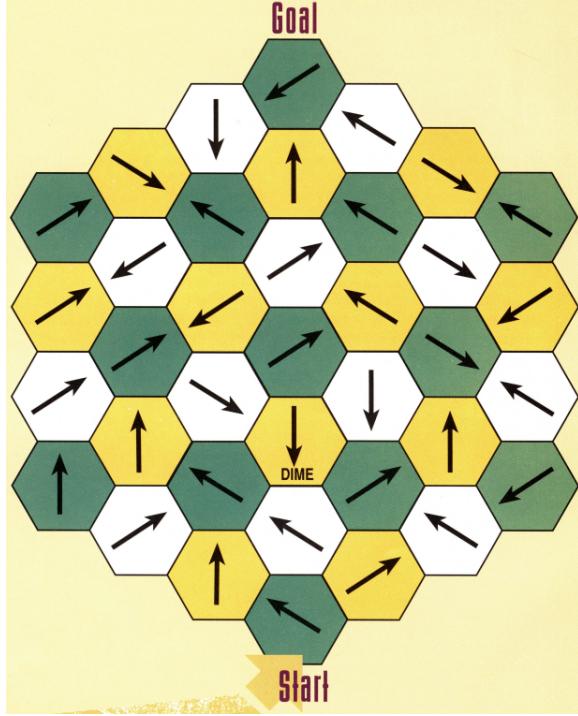


Figure 2.7: Arrow Hockey instance from SuperMazes Abbott (1997).

state graph reduction, especially when it results in consuming a much larger amount of (mostly wasted) space. Unlike direction state-changing mazes, there is very little difference between the separate subgraphs in the arrow hockey maze—the subgraphs would be identical except for the cells that point at the dime and the cell with the dime itself, which would acquire additional edges.

2.2.4 Connections Mazes

Another maze often used in the undergraduate algorithms class is Grandpa’s Transit Map (Abbott, 1990, 18-19). Abbott refers to this type of maze as a “Connections Maze” or an “Either-Or maze” in his work SuperMazes (Abbott, 1997, 28-29). Connections mazes resemble a graph and have vertices (villages) and edges (transit lines). The goal is to travel from the start (Startsburg) to the goal (Endenville). From the starting vertex, one may choose any outgoing edge (in all of Abbott’s connections mazes, the start only has a single edge connecting it to another vertex, so this is a simple extension of his rules) to take to a connecting vertex, but from that point forwards, the edge on which one chooses to exit a

circle must have the same type or the same color (a “free transfer” in Grandpa’s Transit Map) as the edge used to enter the vertex. Additionally, this maze has a no-U-turn rule, meaning that one cannot leave a vertex on the same edge used to enter that vertex, even though it (trivially) has the same color and/or type as the edge used to enter the village.

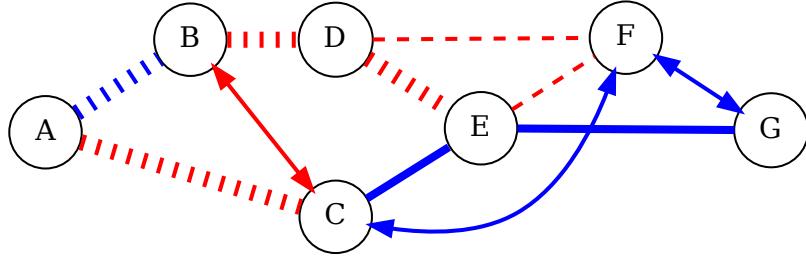


Figure 2.8: Connections maze with two colors and four line types.

In the connections maze given in Fig. 2.8 the objective is to move from the starting vertex (A) to the finish vertex (G). It is possible to move from the starting vertex (A) to either B or C. If one chooses to move to C from A, then the only option (given the rules above) is to take the red arrow line to B, then to D, then the path splits and one could choose to move to either E or F.

In the model for connections mazes, the transit lines in the maze become vertices (two vertices per edge in the original maze) and the villages become edges. Consider an arbitrary connections maze M . Let G be a directed, unweighted graph. For every line L that connects villages v_1 and v_2 in M , create two vertices in G , one which represents “boarding” transit line L from village v_1 and another which represents boarding transit line L from village v_2 . For the vertex that represents boarding L from v_1 , add directed edges to the vertices that represent boarding transit lines at v_2 with the same color or the same type as L (excluding the vertex that represents boarding L at v_2 because of the no-U-turn rule). For the vertex that represents boarding L from v_2 , add directed edges to the vertices

that represent boarding transit lines at v_1 with the same color or the same type as L (excluding the vertex that represents boarding L at v_1 because of the no-U-turn rule).

Because it is possible for multiple transit lines to connect to the starting village A , add a unique starting vertex and connect it to the vertices representing boarding any transit line at village A . Similarly, add a unique finish vertex and add edges from the vertices representing the transit lines boarded at all villages that connect to the finish village to this new finish vertex. The example given in Fig. 2.9 shows the graph representation of the connections maze given in Fig. 2.8. Again, ignore the colors and shapes of the vertices, these will be explained in greater detail in a later chapter.

Traveling one direction along a transit line is completely different than traveling the opposite direction on the same transit line. (these are separate vertices in the state graph representation). This can be used to create clever and tricky mazes that involve retracing many of the same transit lines in reverse order.

2.2.5 Multiplayer Mazes and Variants

Many logic mazes involve multiple players or “pointers” as Abbott refers to them. Some examples from Mad Mazes are Spacewreck, Meteor Storm, and Theseus and the Minotaur (Abbott, 1990, 32-35). In the undergraduate algorithms class, we use Spacewreck, which has only two players, but the model for two players generalizes to mazes with n players.

Spacewreck involves a graph-looking maze with rooms (vertices) and corridors (edges). The corridors can only be traversed in one direction in this variant, and one of the two players must reach the room marked “Goal” to complete the maze. Both rooms and corridors have associated colors. Player 1 starts in room A and player 2 starts in room B. Only one of the two players can move at a time. If player 1 is in a room with color c , then player 2 can move through an adjacent corridor of color c to a new room. If player 2 is in a room with color c , then player 1 can move through an adjacent corridor of color c to a new room.

Consider Fig. 2.10. Players 1 and 2 start in rooms A (red) and B (blue), respectively.

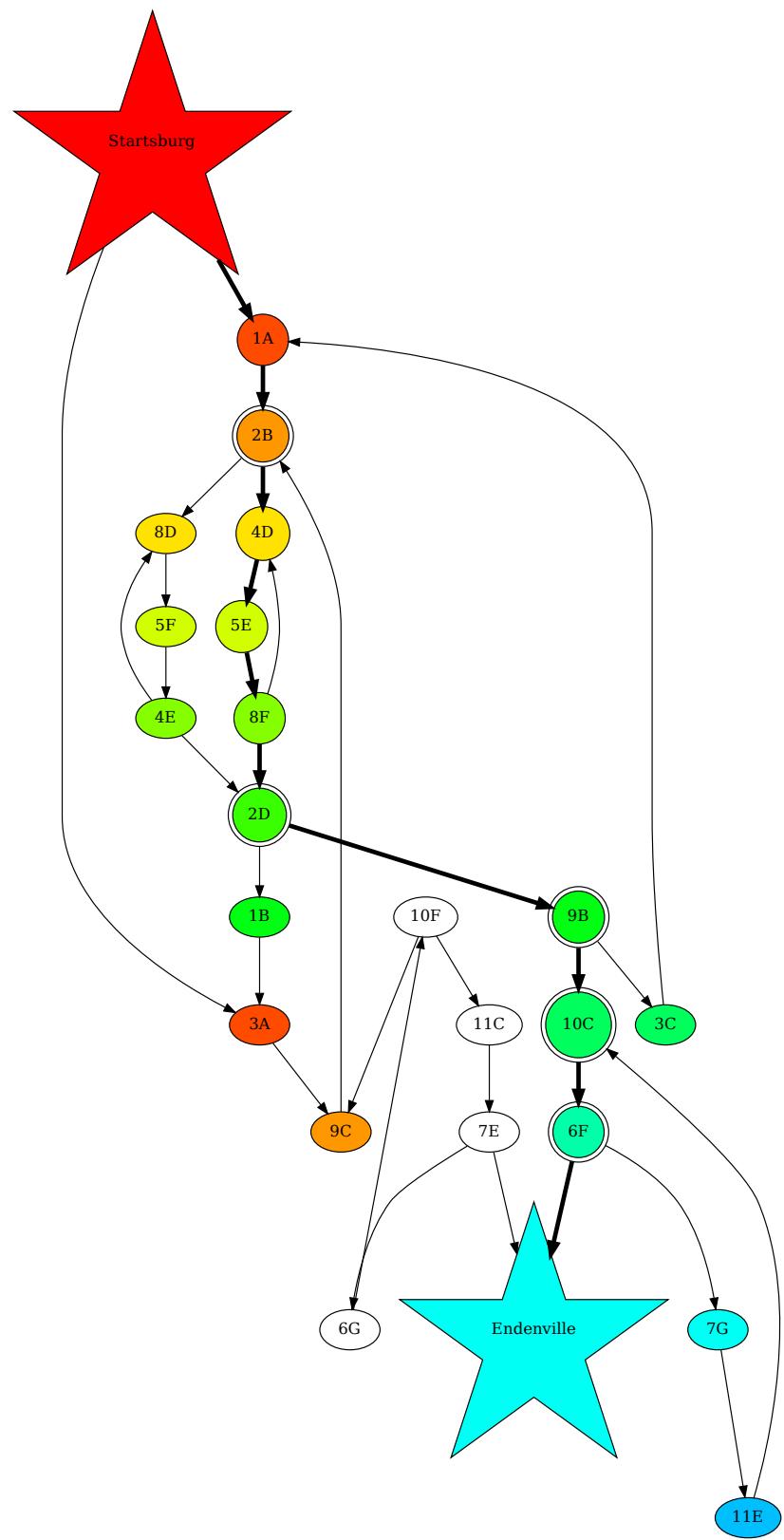


Figure 2.9: Graph model of Fig. 2.8

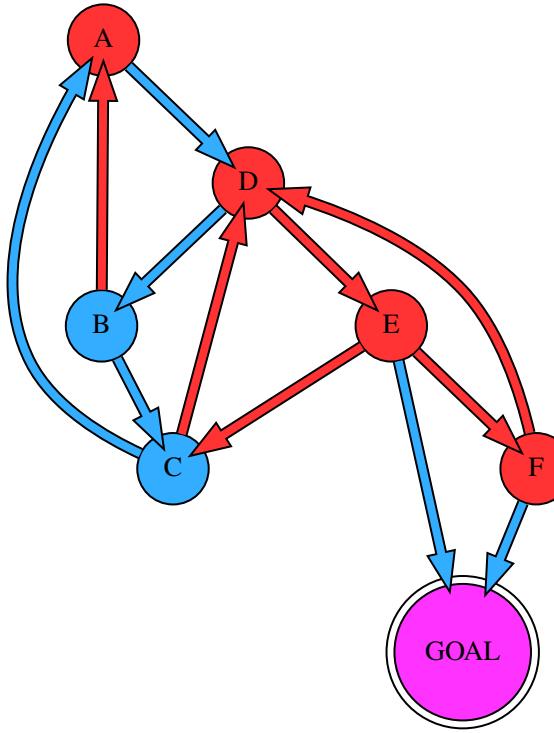


Figure 2.10: An example Spacewreck maze.

Either player 1 or player 2 can move, and they do not have to alternate (one of the two can move down multiple corridors while the other does not move). Player 1 can move down the blue corridor to D because player 2 is in a blue room, OR player 2 can move down the red corridor to A because player 1 is in a red room. This second move results in both players in room A, which is a dead end because A has no outgoing red corridors. A good notation system to keep track of the maze state is the locations of the two players in alphabetical order. Because the players do not have to alternate and only one of them must reach the goal, we do not need to track which player is where. The two players start in position AB. Player 1 moves to D so they are now in position BD; then player 2 moves to A to they are in position AD.

$AB \rightarrow BD \rightarrow AD \rightarrow AE \rightarrow AC \rightarrow CD \rightarrow DD \rightarrow DE \rightarrow EE \rightarrow EC \rightarrow GOAL$ is one possible path to reach the goal.

The state graph model for multiplayer mazes is polynomial in the number of players. All possible locations of each player/pointer must be included as vertices, with edges

representing the movement of one player. In the case of Spacewreck, we use the notation system introduced previously to denote each vertex - AA, AB, AC ... AF BB BC BD ... FF. We give the two letters that represent each player's location in lexicographic order; this helps avoid repetition of vertices, e.g., vertex AB and vertex BA denote the same state in the maze, regardless of which player is in vertex A or B, thus having BA and AB is superfluous. It is easiest to generate the vertices by iterating over all pairs of rooms in the original maze in lexicographic order.

In the rules of this multiplayer maze, only one of the players must reach the goal vertex. Therefore, when generating the state graph, any vertex that has one of the two players in the goal room is the same state, which is denoted “GOAL GOAL” (e.g. “A GOAL”, “B GOAL”, would all be one vertex in the state graph model denoted “GOAL GOAL”).

Let G be a directed, unweighted graph with rooms R . Iterate over the rooms in pairwise fashion to generate all vertices AA, AB, etc. For each vertex, check if the color of room 1 matches any outgoing corridors from room 2. If so, add an edge from the current vertex to the vertex that represents the player moving down that corridor. Do the same process for room 2 and outgoing corridors from room 1. If both players are in the same room, then the same edges could be added multiple times, but this graph representation simply ignores edge-adding if the edge being added is already in the graph (it is not a Multigraph). If the outgoing corridor leads to the GOAL room, then add an edge from the current vertex to “GOAL GOAL.” See Fig. 2.11 below for the graph representation of the Spacewreck instance given in Fig. 2.10.

This model is polynomial in the number of players. It has on the order of n^p vertices where p is the number of players. If there was a Spacewreck variant with three players, then the model would contain vertices AAA, AAB, AAC, etc. This maze has no “turns”, i.e. any player can move at any time instead of alternating. If there were turns; however, this would add another constant factor onto the number of vertices in the graph representation $p * n^p$.

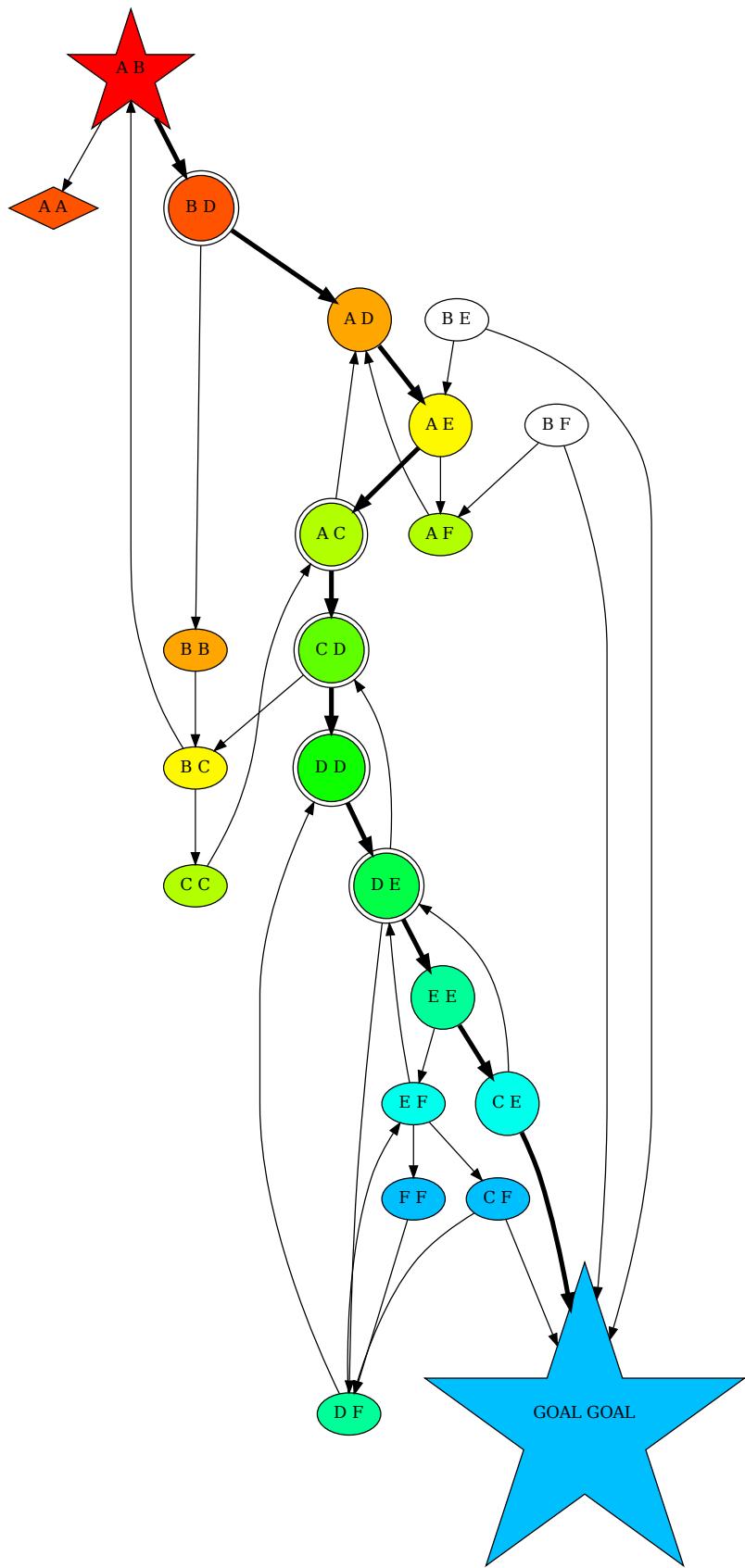


Figure 2.11: State graph model of the maze in Fig. 2.10

2.2.6 Misc. other Mad Mazes

All logic mazes (that we have found) can be modeled as a state graph and solved with a BFS. Theseus and the Minotaur (Abbott, 1990, 34-35) involves a model similar to Spacewreck, with the additional complexity of determining which player's turn it is at each location pair. Alice in Mazeland (Abbott, 1990, 28-29) is on a grid similar to the arrow and jumping mazes, except the arrows point in multiple directions that indicate the allowed directions of movement. The player's step length starts as 1, but certain squares increase the step length by 1 and others decrease it by 1. This model is similar to the state-changing Apollo's Revenge and Jumping Jim mazes - create a separate subgraph for each possible step length, and connect the subgraphs with edges to and from the vertices that change the step length. This model is polynomial in the side length of the grid because the step length can be anything from $1..s - 1$ where s is the maximum side length of the grid. Dice mazes (or Rolling-cube mazes as they are often called) are also played on a grid and have 24 vertices in the graph model per square - six different numbers can be on top of the die, and for each of those six numbers, four different numbers can be facing the bottom (or equivalently, the top) of the grid (Abbott, 1990, 37).

So far all mazes that we have found can be modeled as a state graph using polynomial additional space compared to simulating the maze directly. It is possible that a maze exists that requires exponential space and time to perform this reduction although we have not found one. Rolling-cube mazes in which one must visit a subset of squares exactly once have been proven NP-Complete (Buchin et al., 2007, 8-9), but this is fundamentally a different problem than the mazes we have considered—maze solvers are trying to find *any* path from the start to the goal in our puzzles rather than a Hamiltonian path, which is the case in Buchin et al. (2007).

CHAPTER 3

MAZE CHARACTERISTICS

3.1 General Maze Characteristics

The primary motivation behind this work was to generate smaller logic maze instances that could be given to students as examples to familiarize them with the rules of the maze and also larger instances that could be used to test their implementations.

Because all of these mazes are reduced to a state graph, which is an unweighted directed graph, we can analyze both the underlying graph representation of a maze along with the high-level problem instance when considering the difficulty and desired qualities of a maze. A unique contribution of this work is that as a result of the state graph reductions, an identical objective function can be used when calculating the score of state graphs across all mazes, regardless of type, and then additional qualities can be applied to the higher-level problem instances individually.

A local search implementation resulted in the highest-quality mazes. This is discussed in detail in Chapter 4, but the main purpose of this chapter is to explain and highlight important maze qualities that one can use in an objective function for local search.

Neller et al. (2011) discuss some desired maze qualities that will be introduced, along with additional characteristics found in our original work. Abbott also mentions several metrics by which he designs his mazes, and those will be mentioned as well. There are some additional maze qualities that we found to be difficult from personal experience, but this has not been verified with human testing. None of the mazes presented (to the best of our knowledge) have been used in a study to determine their true difficulty, instead we have picked metrics from Abbott, Neller et al. (2011), and personal experience.

Before moving into specific desired maze qualities, it is important to note that when designing a maze for human beings to solve, a common solving method is to attempt to work backwards from the goal instead of forwards from the start (Abbott, 1999, 38). Therefore, it follows that the maze ought to be equivalently difficult when attempting to

solve it forwards or backwards, or it will be easily solved through back-tracing.

Considering not only the state graph, but also its transpose, is important when considering the following maze characteristics. From experience, when calculating the value of any maze metric, a useful method to enforce the same difficulty in the transpose is to calculate the value of the metric on both the state graph and its transpose, and take the minimum (maximum) value and add that to the objective function, assuming one is trying to maximize (minimize) the objective function. For the remainder of this chapter, it is assumed that maximization of an objective function is the goal.

Fig. 2.9 is the state graph of Fig. 2.8 based on the reduction given in Chapter 2. Below is an explanation of the colors and shapes in the state graph diagram.

- Shapes
 - The start and goal vertices are star-shaped.
 - Diamond-shaped vertices are unreaching, that is, it is impossible to reach the finish from them.
 - Vertices on the shortest path (only one path if multiple shortest paths) are circular shaped.
 - *Required* vertices are double-circled. This means that *any* path from the start to the finish *must* pass through these vertices.
 - Edges between vertices on the shortest path are bolded for convenience.
- Color
 - Vertices are shaded according to the distance from the start vertex.
 - The more red the color of a vertex, the shorter the distance to reach it from the start vertex.
 - Vertices colored similar to the goal vertex are at about the same distance from the start as the goal vertex.
 - Vertices colored a darker blue than the goal vertex are at a further distance from the start than the goal vertex.
 - Uncolored vertices are unreachable from the start.

3.1.1 Paths, Shortest Paths, and Number of Nodes

The first indicator of maze difficulty is the number of vertices in the graph representation. Given two mazes with similar characteristics, the one with additional vertices will be more difficult to solve simply because increased vertices enables larger holes/traps (explained in a later section), longer true and false paths (the true path leads to the solution while a false path does not), more decisions are made, etc. That said, directly increasing the number of vertices without thought or randomly does not directly increase the difficulty of the maze; it is these additional vertices contributions to other aspects of the maze that makes it more difficult, thus it should not be included as a direct term in an objective function. However, given two mazes generated with the same objective function that have about the same characteristics, the maze with additional vertices will be more difficult than the alternative. The number of vertices in the state graph can be used to weight terms in the objective function to ensure that it is accurate relative to the size of the maze.

What portion of the vertices should be involved in the shortest path? According to Abbott, it is best to leave a substantial amount of space for several long false (Abbott, 1997, 27). From experience with our program, a good number is 20-40%, but no more than 40% to leave enough space in the maze for long false paths and other characteristics that increase the difficulty of the maze. If a majority of the maze is involved in the shortest path, then over 50% of vertices are “correct” and cannot be used for other traits to increase the difficulty of the maze. When including this term in the objective function, if the shortest path is over 40% of the vertices, then a deduction is applied to the score.

Beyond the length of the shortest path itself, what about multiple shortest paths? Neller et al. (2011) find that there is a level of satisfaction achieved when one has found the shortest, or best solution, and the existence of a unique shortest solution can motivate solvers to continue working on a maze to improve their solution (Neller et al., 2011, 193). Thus we apply a deduction to the score if there are multiple unique shortest paths (with

some exceptions, see specific maze qualities below).

3.1.2 Branching

Branching is a relatively intuitive characteristic, and it is similar to the idea of a “branching factor” in the growth of a search tree/space. Branching simply asks, after exactly X moves from the start, in how many different locations (vertices) could one be? This helps to ensure that there is not a long section of forced moves at the beginning (or at the end), at which point the maze might as well start after the forced moves. It also helps to confuse solvers working both forwards and backwards if one could be in 14 different possible locations after only three moves when moving either forwards from the start or backwards from the finish.

When calculating branching, one must choose how many possible moves to consider. An effective method to calculate a branching score is to vary the number of moves that are considered from the start position based on the length of the shortest path. The longer the shortest path, the larger the values of X when calculating the total number of possible locations after X moves from the start. 10-15% of the shortest path length is a good starting point.

An additional factor to consider with branching is if repeats will be considered (e.g. there is a vertex that can be reached after either 2 or 3 moves from the starting vertex). Generally, vertices should not be double counted when considering branching because a human maze solver that is less than 10% into a reasonably sized maze will likely notice that they are entering a state that they could have reached from the start in less moves, and thus not make that choice. It is possible to consider repeats, but branching scores with repeats tend to disproportionately impact the overall maze score.

3.1.3 Reachability

A *reaching* vertex v is a vertex from which it is possible to reach the goal g . A *reachable* vertex is a vertex that can be reached from the start s Neller et al. (2011). A traversal from the start (finish) on the state graph (transpose) can be used to calculate the

portion of reachable (reaching) vertices. 60% is a good minimum portion of the maze to be reachable (reaching) from the start (finish). This restriction can heavily influence the traps that one creates, as many traps are often only reachable when moving in one direction through the maze.

Reachability also measures the efficiency of a maze because the number of vertices in the state graph is a function of the puzzle size, which usually does not change while generating an instance. Often, one chooses a size (say, 30 rooms and 72 corridors in a Spacewreck maze) and the generator does not modify the size, it merely changes the other properties of the maze (colors, which rooms the corridors connect, etc.). Using the formula for combinations with replacement with $n = r - 1$ and $k = 2$, and then adding one for the goal, a state graph of a Spacewreck maze with r rooms will have $\binom{r}{2} + 1$ vertices. A 30 room instance will have 436 vertices. If only 200 of the vertices were reachable and/or reaching, this would be a poor use of resources.

Vertices that are both unreachable and unreaching are denoted *isolates*, and these represent wasted resources—vertices in the state graph that cannot be reached from the start nor backwards from the finish. The puzzle instance ought to be redesigned so these are included as a part of the maze. Isolates negatively influence the score of the maze in objective functions as a result.

Depending on the type of logic maze, if bidirectional movement is possible and the state graph is undirected (similar to a classic maze), then all vertices that are reachable will also be reaching (even if one must pass through the start to reach them in the opposite direction or vice versa). There will only be isolates and vertices that are both reachable and reaching.

3.1.4 Cycles, Holes, and Whirlpools

From the definitions of reachable and reaching vertices, we can define several different traps to confuse and hinder the maze solver. A dead end of a maze is a set of reachable, unreaching vertices in the state graph. A dead end can consist of one or many vertices. A

reverse dead end is a set of reaching, unreachable vertices in the state graph.

A *black hole* is a set of strongly connected, reachable, unreaching vertices in the state graph. In effect, it is a false path that ends in loop(s) instead of at a singular dead end. Black holes, especially large black holes, can make a maze very difficult to solve, because the moment one enters it, it is no longer possible to reach the finish, but a maze solver may spend a lot of time in the trap before realizing there is no escape. This definition is slightly different than the one given in Neller et al. (2011), because we wanted to focus on the strongly connected vertices, which is the core of the trap, and neglect the fringes. These significantly increase the difficulty of a maze because a solver can travel far through a black hole (especially a large black hole) before realizing they are retracing their steps and there is no path to the solution. Black holes usually lead to one restarting the maze after entering and moving through one—usually the solver does not remember how they initially entered the black hole, so they end up restarting the maze (Neller et al., 2011, 193).

A *white hole* is a set of strongly connected, reaching, unreachable vertices in the state graph. It is identical to a black hole when considering the transpose of the state graph (with the start becoming the finish and vice versa). Only maze solvers working backwards from the solution can find themselves in white holes. Again, these increase the difficulty of a maze and help confuse backwards-minded maze solvers, but not to the extent of black holes. Fig. 2.9 has a small white hole: the vertices 10F, 11C, 7E, and 6G make up a four vertex white hole.

Generally, we have found that black holes make a maze more difficult than white holes simply because if the maze cannot be easily solved in a brute-force manner by moving forwards or backwards, most maze solvers will use analytical/systematic techniques to solve the maze. These techniques are almost always applied to the maze when solving forwards from the start rather than backwards from the finish.

It is important to note that these type of traps only effect one direction of solving the maze. Reverse dead ends and white holes do not affect individuals solving the maze moving forward from the starting vertex because they cannot be reached when moving

forwards. Similarly, dead ends and black holes do not hinder solvers attempting to move backwards from the solution. These traps are in direct conflict with the reachability score, because their existence implies that there are vertices that cannot be reached when moving in both directions. However, the state graphs of many of these puzzles, especially if the puzzles are reasonably sized, are sufficiently large (100+ vertices in the state graph) that the increase in difficulty from the traps is worth the sacrifice of reachability in one direction for a subset of vertices.

A *whirlpool* is a set of strongly connected, reaching, reachable vertices. In effect, it is a trap that can be reached in both directions, and the placement of such traps is more difficult and important. A whirlpool that is close to the start is vulnerable to back-tracing from the finish, and the same can be said of a whirlpool close to the finish. This is because one can often “see” a path to the goal state when within 5 moves of it, even if one passes through a large trap in these final 5 moves.

When considering maze difficulty, it is always preferred to have some sort of hole instead of a dead end because holes are naturally more troublesome than a simple dead end that halts all progress—one often wastes a lot more time moving around a hole than stopping at a dead end, and additionally, at simple dead ends one can usually retrace their path backwards before that decision was made. After spending time cycling around a hole, often one must restart the maze because the solver does not remember how they initially entered the trap.

It is possible to have too large of black/white holes that make one direction of movement much easier than the other. Typically reachability constraints prevent a maze from having a single large hole that can be easily solved through back- tracing and vice versa, and also applying the method discussed above (taking the minimum of the black and white hole scores before adding onto the objective function) results in a balanced maze with sizeable black and white holes, or neither.

If the logic maze has bidirectional movement (such as Meteor Storm (Abbott, 1999, 33)), then black holes and white holes cannot exist, there will only be whirlpools for this

type of trap.

3.1.5 Decisions, Required Vertices, Bridges/Dominance

The presence of traps such as black/white holes and whirlpools does not necessarily imply that a maze is difficult. We have found many mazes where multiple traps exist, but a solver does not encounter such traps unless they are unlucky, or, alternatively, a lucky solver would never encounter such traps.

Intuitively, we want there to be as many chances as possible for a solver to become lost in these traps, or they do not truly serve to increase the difficulty of the maze. In fact, we want to *ensure* that this is the case. The first solution that comes to mind is to consider all of the *decisions* that a maze solver must consider at each step along the shortest path. Consider each vertex on the shortest path n . From n , there are several cases for each immediate descendent d (the resulting vertex of each outgoing edge of n):

- d is the next vertex on the shortest path.
- d is in an unreaching trap (black hole/dead end).
- d is in a reaching trap (whirlpool) or on a suboptimal path to the solution.
- d is a previous vertex on the shortest path.

We would like there to be as many decision points as possible that lead into reaching/unreaching traps or suboptimal paths along the shortest path. However, there are several issues with merely counting the number of outgoing edges from each vertex on the shortest path that satisfy this requirement. The simplest problem would be (depending on the maze) the decisions within a few vertices of the solution vertex are often irrelevant because when one is only a few moves away from the solution, it is clear by inspection what decisions to make. This can vary a bit from maze to maze. For instance, in the connections mazes shown previously, it is easy to see the path to the solution when one is within 3-4 vertices of the goal. In the jumping mazes; however, it is not so obvious and one may need to be within 1-2 vertices of the solution to see the path to it. When creating an objective function that considers decisions, it is up to the designer to choose the move cutoff when determining the value of decisions that are factored into the score of the maze. However,

the mazes presented in this work are at a level of difficulty such that most if not all solvers will attempt back-tracing, which means that they will be at least somewhat familiar with the paths of length 3-5 working backwards from the finish. Not considering decisions for vertices within 5 moves of the solution is a good rule of thumb to use in this context.

Another consideration is that not all of these decision points should be weighted equally. If one edge leads into a 5 vertex black hole and another into a 150 vertex black hole, the latter should modify the score more than the former. There are several ways to handle this which will be discussed further in the next section.

If there are multiple shortest paths, or multiple paths to the solution in general, not all solvers may have to consider every decision that is factored into the score, which is a significant issue. Decisions are one of the primary factors in determining the difficulty of a maze (Neller et al., 2011, 194). Thus, factoring decisions into the score of the maze that not all solvers may have to consider could falsely increase a maze’s perceived difficulty. The natural solution is to consider only the decisions that *every* solver, regardless of path chosen, must make. This leads to the following question: given a start vertex s and a goal vertex g , considering all of the paths from s to g , which vertices are in every path? In other words, what vertices are “required” vertices R that must be present on *all* paths from s to g . Equivalently, if any vertex in R is removed, a path no longer exists from s to g .

This question has been studied and solved in theory related to control-flow graphs, and is referred to as *Dominance*. A vertex v *dominates* another vertex u if v lies on every path from the entry vertex to u .

We need to determine which vertices dominate the goal vertex with an entry node of the starting vertex. Let D denote the vertices that dominate the goal with an entry point of the start vertex. If a vertex $n \in D$ (n dominates the goal vertex), then any dominator d of n also dominates the goal vertex, that is, $d \in D$. Thus, we can consider a *dominator tree* of all the vertices that dominate the goal vertex. We calculate the dominators of the goal vertex by calculating the immediate dominator of the goal vertex id_{goal} and then the immediate dominator of id_{goal} , etc., until reaching the start vertex. The immediate

dominator is the parent of a vertex in the dominator tree.

Dominance has been used to show the safety of code-reordering operations and can be used in control flow graphs to determine which lines of code must execute before others. Cooper et al. give a $\mathcal{O}(V^2)$ algorithm that in practice runs faster on graphs with less than 1000 vertices than the classical $\mathcal{O}(E \log(V))$ Lengauer-Tarjan Algorithm (Cooper et al., 2006, 1-2). State graphs of typical mazes do not usually exceed this number of vertices. After computing the dominator tree, we can determine which vertices dominate the goal vertex. These vertices (specifically their outgoing edges) define the decisions that all maze solvers will encounter, and we can choose to include only the decision score associated with these vertices.

3.1.6 Long False Paths (Decisions continued)

Abbott directly mentions leaving space for what he denotes “long false paths” repeatedly in his work on designing mazes (Abbott, 1997, 26-27), (Abbott, 1990, 36-40). A false path is a path that does not lead to the goal. Holes, whirlpools, and forks from the solution path that waste time are all considered false paths.

Long false paths correspond well to suboptimal decisions at the required vertices. We would like use the same scoring method for all decisions. In the event the solver makes a suboptimal decision, the degree of the incorrectness of the decision should factor into the decision score of that decision. When a suboptimal decision is made, there are three cases possible for the resulting vertex d .

1. d is a previous vertex on the shortest path.
2. d is in an unreaching trap (black hole/dead end).
3. d is in a reaching trap (whirlpool) or on a suboptimal path to the solution.

The score associated with the first case should be minimal. Many solvers will remember their previous 10-20 moves, so a decision that leads backwards on the correct path to a vertex that has likely already been encountered does not make the maze particularly more difficult. Even if the shortest path is of length 80 and a vertex 50 moves into the path has

an edge to the 5th vertex on the path, a solver is likely to recognize that vertex is close to the start and not make that choice.

In the second and third cases, the score associated with the decision is how long the solver can move through the trap without “making progress.” In the case of the unreaching trap, it is relative to how long before the solver realizes they are in a trap going around in circles and need to restart. Algorithmically, this asks what is the maximum distance a solver can move in the trap without retracing their steps? Once they start retracing their steps in the trap, they will realize that they are stuck in a trap and need to find a way out of it (probably by restarting the maze).

In the third case, the solver has entered a reaching trap, and the score is calculated in the following manner: what is the furthest distance the solver can move without getting closer to the solution than the vertex at which they made the suboptimal decision and without retracing their steps further back along the shortest path? For example, consider that a solver is at a vertex on the shortest path that is 20 moves away from the goal. They make a suboptimal decision and move to a vertex that is 40 moves away from the goal. The maximum score that could be added as a result of this decision is $40 - 20 = 20$.

Based on the discussion in the previous section, we will apply these decision scoring criteria to only the required vertices, that is, the vertices that dominate the goal vertex, and we will not include required vertices that are 3 moves away (or less) from the goal vertex.

3.2 Mad Maze Specific Characteristics

Mazes often have characteristics unique to the problem instance itself that ought to be included in the objective function. Because the state graph abstracts away some qualities of the maze instance, these characteristics must be included in the objective function of each specific maze type.

3.2.1 Color

Many logic mazes (such as Connections or Spacewreck mazes) include multiple colors, and we desire even color distributions. If a Connections or Spacewreck maze has four colors, it is clearly undesirable for 90% of the rooms/villages and corridors/transit lines to all be the same color. This can be enforced by counting the number of appearances of each color and deducting from the score the absolute value of the difference between the minimum number of appearances of a color by the average. One can also subtract the difference between the mode color and the average, and also square or cube this difference if necessary. In Spacewreck mazes specifically, having an even distribution of color often leads to additional isolates because it restricts possible moves, so the designer must choose which is of more importance.

3.2.2 State Changes

Many logic mazes involve state changes. In the jumping and arrow mazes introduced in this paper, changes in the movement direction qualify as a change in state. In these mazes, we want to maximize the number of state changes that occur along the shortest path (or perhaps maximize the number of state changes between successive vertices in the dominator tree). It would be undesirable for one of these mazes to not require the solver to change the movement direction between cardinal/diagonal and forwards/backwards. Thus, one can count the number of state changes along the shortest path/required vertices, and add the product of this quantity and a multiplier (5-10 is usually sufficient) that determines the additional score that results from a state change.

However, this can result in mazes with a large majority of the grid squares being circled (all of them in some cases), which is suboptimal.

6	1	4	4	3	3	2	1	2
7	3	2	1	6	6	6	1	
1	5	2	2	3	2	5	2	
2	1	4	2	3	2	3	3	
2	6	2	4	4	4	1	4	
3	4	1	2	1	4	2	2	
1	5	1	4	1	4	3	1	
5	6	4	4	4	1	4		GOAL

Figure 3.1: A jumping maze generated by a function that maximizes state changes without a limit on the number of circled squares.

To have both state changes along the required vertices and to ensure that an optimal portion of the grid squares become locations of state changes, we choose a desirable portion of the grid squares (20% is a good number) to be state changes. Then calculate the difference between the actual number of state-change grid squares and the desired portion. Square (or cube) this difference and subtract it from the score in the objective function.

3.2.3 Revisiting Locations and Doubling Back

In all of the mad mazes presented in Chapter 2, it is possible to revisit the same location in the maze multiple times. This needs no explanation in jumping and arrow mazes; in Spacewreck mazes because state is determined by the locations of both Lucky and Rocky, one player can revisit the same location (say, A) multiple times in a completely separate part of the state graph. This can be disorienting and confusing for maze solvers because it feels like they are retracing their steps even though they may not be, which can make a maze significantly more difficult to solve. In the Spacewreck maze instance given in Mad Mazes, Abbott has designed the maze so the solver must trace around one initial loop (A-J-O-N-H-D-A) two times with one player and another loop *three* times with the other (B-G-K-F), and only one exit from these dual loops leads to the solution (Abbott, 1990,

37).

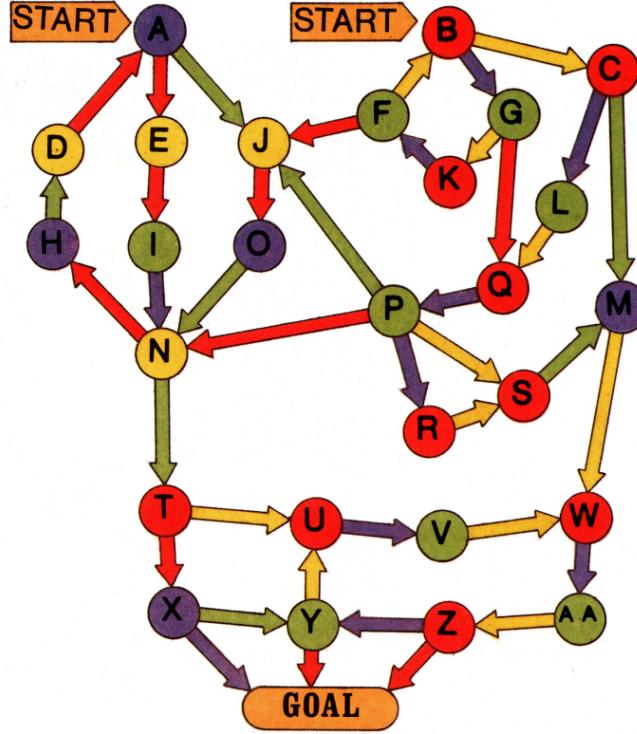


Figure 3.2: The Spacewreck instance given in Mad Mazes (Abbott, 1990, 32).

The concept of “state” that logic mazes introduce enable this revisit, which Abbott uses in many of his mazes to confuse the solver (Abbott, 1990, 36). In the case of Spacewreck, we can count the number of vertices (required or on the shortest path) after both players have moved out of their starting position where either player returns to one of the starting rooms. In connections mazes, we can keep track of the visited villages and count the number of times a village is revisited. An especially tricky aspect of connections mazes is that traveling one direction on a transit line is an entirely separate vertex in the state graph than traveling the other direction on the line. This idea enables one to create entertaining mazes where one traces a large path out into the maze, makes a U-turn in some fashion, and follows the same exact transit lines back near to the starting village and then follows a short path to the finish that is now possible. Abbott uses this idea in his Grandpa’s Transit Map instance.

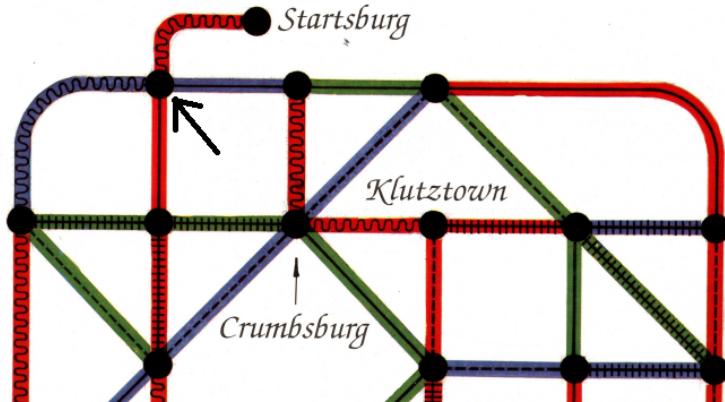


Figure 3.3: The top section of Abbott’s Grandpa’s Transit Map connections maze (Abbott, 1990, 19). The arrow points to the village mentioned when discussing transit lines exited from the start.

The solver begins the maze at Startsburg on the red curvy line, and can transfer to either the blue curvy line to the west or the red straight line south at the next village intersection (indicated with a black arrow in Fig. 3.3). However, any path to Endenville (the goal) *must* return to this same village intersection and take the blue straight line to the east, the one line it is not possible to transfer to when entering the maze from Startsburg. Thus, any solution must head out into the maze, make a U-turn, and follow the same transit lines back to this village so the blue straight-line transit line can be boarded at this village and taken east.

The one caveat of this tricky maze instance is that there will always be two possible shortest paths—the transit lines that one uses to make the U-turn can be traversed in both directions, so this needs to be factored in when considering the score of having a single shortest path. We have found that for this type of maze, it is preferred to allow for two or less shortest paths so these U-turn type instances can be generated.

In jumping mazes, specifically with cardinal/diagonal state changes, it can be especially desirable to re-visit the start square along the shortest path (or as a required vertex) but in the diagonal state instead of the cardinal state. Typically, a modest increase to the score equal to the number of squares in the grid is a good amount to reward for this occurring. Note that this cannot occur in arrow mazes because the starting square arrow must point

onto the maze, forcing its tail to point off of the maze.

The concept of doubling back is similar to revisiting, except that it is more unique to the grid mazes discussed (arrow and jumping mazes) because it involves a disorientation of the solver not from revisiting the same square in multiple states, but rather multiple repeated movements in the same row, column, or diagonal. Abbott uses this concept in many of his grid mazes. In the Apollo's Revenge Maze, the shortest path moves up and down a column, followed by a reverse of direction, and then moves up and down the same column (Abbott, 1990, 36). Given the shortest path on a maze (or adjacent required vertices), it is possible to calculate the amount of moves in a maze that require doubling back, moving either up and down the same column repeatedly, or along the same diagonal. The jumping maze given in Fig. 3.4 maximizes the use of doubling back.

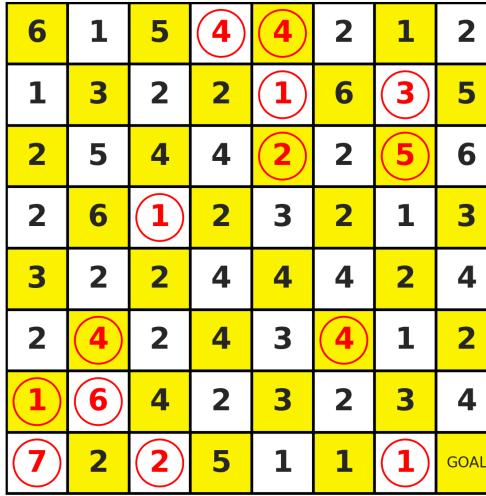


Figure 3.4: Jumping maze with an abundant use of doubling back.

The maze given in Figure 3.4 begins at (1,1). The path to the goal then moves south to (7,1), northeast to (6,2), east to (6,6), and then doubles back for the next 6 moves along the main diagonal: (6,6) to (2,2) to (5,5) to (1,1) to (7,7) to (4,4) to (6,6). This amount of doubling back along the main diagonal will confuse many mazes solvers, and it also brings a degree of surprise and enjoyment to solving the maze not present in other mazes without

this novel feeling of doubling back.

3.3 Component Graph

As mazes grow, it becomes more difficult (nearly impossible) to identify the characteristics of the state graph by inspection. This is where it becomes useful to generate the component graph, or *condensation* of the state graph, which makes it substantially easier to see traps and the overall structure of the state graph G . The condensation of G is the graph with each strongly connected component contracted into a single vertex. After generating the condensation, edge labels are added to show the number of *inter*-edges between the components in the original graph, and self-loop edges are added to show the number of *intra*-edges in each component: edges that are between vertices *within* each component. The same coloring and shape rules are applied as explained previously in the complete state graphs.

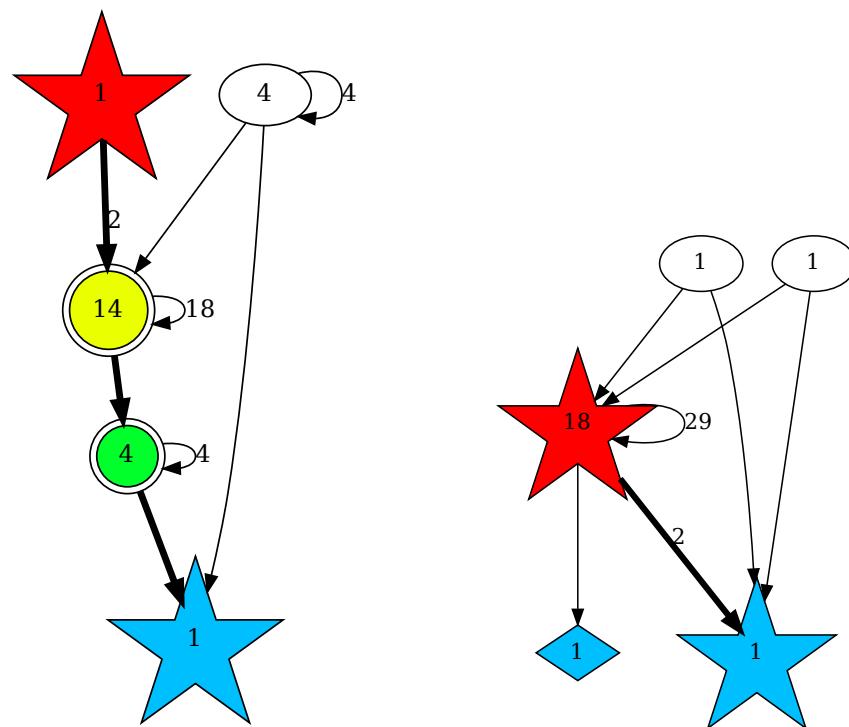


Figure 3.5: Component graphs of the connections and spacewreck mazes given in Figures 2.8, 2.9, 2.10, 2.11.

In the first maze given in Fig. 3.5, we can see that we pass through two small strongly connected components, or two smaller mini-mazes, with a 4 vertex white hole. The second maze component graph does not tell us a whole lot, other than the maze solver starts within an 18-vertex strongly connected component with 29 self-edges and 2 possible ways out (edges) that lead to the solution vertex.

While these component graphs are manageable, they are for trivial maze instances. Larger mazes such as the jumping maze in Fig. 3.4 or the Spacewreck instance in the maze book Fig. 3.2 have component graphs that are too unwieldy and large to be shown in this paper due to a large number of single-vertex strongly connected components that are unreachable/unreaching dead ends. Removing all single-vertex strongly connected components from the condensation that are unreachable *or* unreaching that do not have both incoming and outgoing edges leads to component graphs that can be shown in this paper. The simplified component graph for Fig. 3.4 (the doubling back jumping maze) is given in Fig. 3.6.

We can see the major components of the doubling back maze in its component graph in Fig. 3.6. The maze solver starts the puzzle in a 40 vertex strongly connected component with 68 intra-edges, with exactly one edge that leads out from this trap to the goal vertex. There is a 30 vertex black hole with 15 direct entrances (edges) from the starting scc into this hole and additional entrances that pass through other unreaching sccs. There is also a 25 vertex white hole with 5 direct entrances and around four times that number of indirect entrances through other unreachable sccs. We can see that the main difficulty of this maze comes from the single edge that leads from the starting scc to the goal combined with the two sizeable holes.

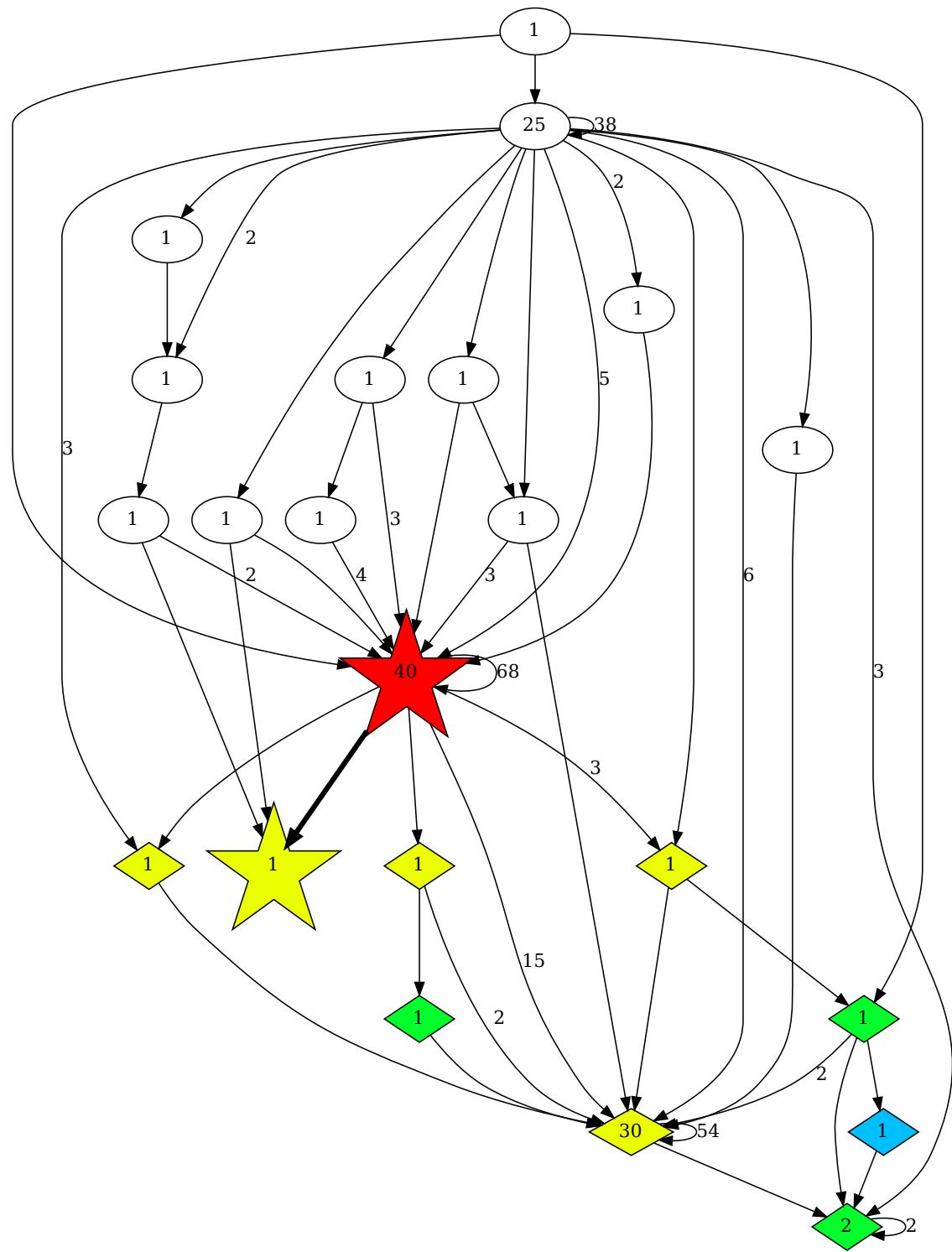


Figure 3.6: Component graph of Fig. 3.4 with single-vertex unreachable or unreaching sccs removed that did not have both incoming and outgoing edges.

CHAPTER 4

IMPLEMENTATION AND SEARCH

4.1 Implementation

Generation of mad mazes lends itself well to an object-oriented implementation. Many logic mazes share similar characteristics, such as being played on a grid or color constraints. The root object contains the state graph logic because all mazes have a state graph representation. More specific types of mazes can either inherit from this object or have a component-based architecture where they add components that they use, such as a grid, colors, etc. See Fig. 4.1 for a UML representation of the architecture that we used for this implementation.

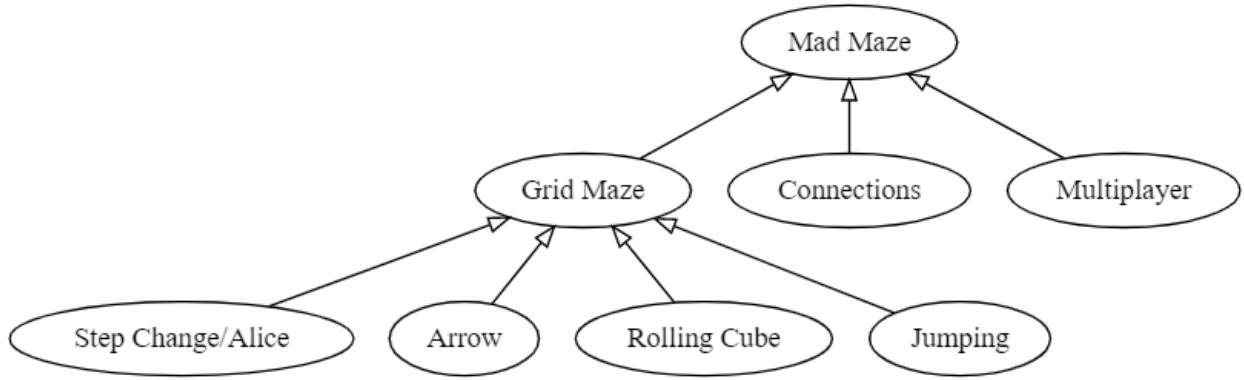


Figure 4.1: Potential inheritance scheme

The root logic maze object contains the state graph objective function while the objects that pertain to each specific maze puzzle call their parent's objective function to rate the state graph, and themselves have operations to consider the attributes of the higher-level puzzle instance (such as an even distribution of colors). The root also contains the state graph visualization and comparison functionality, state graph scoring and analysis, and general (abstract) methods for reading input, outputting solutions and generation that can be overridden by its children.

4.1.1 Program Infrastructure

There are several ideas that can be used to efficiently implement the detection of attributes discussed in Chapter 3. Firstly, the reduction to state graph combined with the use of a robust graph library enables one to use pre-implemented versions of all the functions necessary to detect and score state graph attributes (dominance, traversals, distance calculations). Python 3 was chosen because the library networkx is an excellent choice that contains all the necessary algorithms to rate state graphs combined with visualization capabilities (exporting to graphviz, which is a good tool for visualizing graphs). Networkx nodes can be any object, with attributes that the user defines. A small portion of preimplemented algorithms are traversals, immediate dominators, shortest path lengths, flow, descendants, ancestors, bfs/dfs trees, etc. See the networkx documentation for details (Hagberg et al., 2008, 11).

Functions that operate on mazes to rate attributes (beyond the graph library functions) should be implemented as helper functions that are given the maze instance and the maze start and finish. This enables reuse of the same function when considering both the forward and the transpose of the state graph by passing in the transpose, the finish as the start, and vice versa.

4.2 Fully Random Generation

A purely random generator was attempted for each logic maze. This is the equivalent of choosing the size of the maze and randomly generating the characteristics of each location/edge in the maze (grid number, arrow direction, color, type). Overall, this method produced unsatisfying results. In the vast majority of instances generated in this manner, there was not a path from the start to the finish. Even in cases where a solution path existed, it was usually a trivial path (short/easy to see), and large portions of the maze were unreachable and unreaching, i.e. wasted space as mentioned in Chapter 3. We omit details on this generation type because the results were disappointing and unfruitful.

4.3 Intelligent Random Generation

These methods of generation do have random components, but are generally much more intelligent than the fully random generation — we focus on improving certain aspects of a maze rather than generating an entire puzzle instance from scratch.

4.3.1 Automation of manual generation

Abbott mentions and discusses several methods of maze generation in both Mad Mazes and SuperMazes. Besides focusing on the traits (long false paths, decisions, etc.) introduced in Chapter 3, Abbott gives more direct instructions on generating a rolling cube maze in SuperMazes.

He specifically mentions first laying out the “true” path, or the solution path through the maze from start to finish. This true path should not take up the majority of the maze to leave room for false paths; Abbott directly mentions that a maze “won’t be interesting” unless it has several complex false paths. After completing the true path, Abbott recommends laying out several long false paths. Then it is a matter of completing the maze: choosing numbers for the remaining squares that create the “most and longest false paths”, i.e., as much confusion as possible (Abbott, 1997, 27).

Our first attempt at “intelligent” random maze generation was to automate algorithms such as the one introduced in the previous paragraph in SuperMazes, and we first tried this for jumping mazes with a diagonal state change. The size of the grid was pre-determined; we used the size of Jumping Jim’s Encore, an 8×8 grid, as a starting point.

Inputs: Minimum shortest path length, Start, Finish, Probability of circling.

Output: A jumping maze instance with circled squares for diagonal state changes.

1. True path generation: The truth path was generated using a backtracking search. A randomized traversal was conducted from the starting point to generate the true path through the puzzle.
 - a) When choosing numbers for squares in jumping puzzles, the numbers must be sufficiently small that it is possible to move in both the cardinal and diagonal

states.

- b) The random numbers that ensure movement is possible are iterated over, then all reachable square locations (children) after one move are calculated.
- c) These children are randomized then iterated over.
- d) We randomly determine if this square will be circled, then loop over all possible numbers (in a random order) that ensure there are reachable locations from this square in both cardinal and diagonal movement.
- e) If the number we have chosen to put onto a square would create a new shortest path to the previous square or a path to the solution that is shorter than the minimum desired length, then that number is skipped. If all numbers would cause this to happen, then we change the square to circled (assuming it was uncircled) and vice versa, then attempt all possible numbers again. If it is still impossible to add a number to this square, then we backtrack to the parent of the current square and try adding a different child to the solution path. If we cannot add any of the current children to the solution path, then we try a different number for the parent.

We have a backtracking solution for generation of the shortest path. The worst case complexity of this algorithm is certainly exponential and it would never finish in some instances, but in practice it quickly terminates.

2. False paths

- a) To attempt to create false paths from the start and the finish as Abbott mentions, we calculate all squares that can be reached from the starting point but are not on the shortest path and do not yet have numbers. Then, we attempt to number these squares and then the subsequent children in such a manner that the shortest path is not intersected. We do this both moving forwards from the starting point of the maze and backwards from the finish of the maze, considering distances of five from the start/finish.

After creating these false paths, we attempt to number the remaining squares using the possible number and circle combinations that would not create a new shortest path to the finish.

While this algorithm resulted in mazes that were certainly improved from the randomly generated mazes, the inconsistent run times and variability in the difficulty of results was undesirable. It was at this point that the authors found the paper by Neller et al. (2011) that recommended the use of local search with an objective function that highlighted maze qualities, and we found this method to be much more successful.

4.3.2 Local search

As mentioned in previous chapters, this type of maze generation does not start from scratch but rather takes a currently generated maze instance and makes small modifications that increase the value of an objective function. When defining local search solutions to problems, there are generally three items to define:

1. An objective function that returns a score given a solution instance to the problem.
We seek to either minimize or maximize this function.
2. A notion of a neighborhood, or a small change made to a given solution to turn it into another solution then rated by the objective function. Generally, there are multiple neighbors of any single solution node.
3. A search function/algorithm, that is, a method by which we choose between the neighbors of a given node in the solution space.

In the context of maze generation, the objective function will be as defined in Chapter 3 and contain metrics to rate both the state graph and the maze instance itself. The neighborhood definition turns one maze instance into another, which usually involves making a small change to the maze instance that is simultaneously reflected in the state graph. We considered generating an ideal state graph and turning it back into a maze, but it is difficult and challenging to determine the minimum requirements for a state graph to be able to map it to a maze instance.

Consider a jumping maze with diagonal state changes. If we take an arbitrary state graph and try to determine if it is a jumping maze state graph with diagonal state changes, there are several requirements that we must have. In effect, we will determine if it is possible to map the state graph onto a grid.

1. We must be able to partition the state graph into two equal-sized partitions (except for the solution/goal, which should have no outgoing edges) such that all edges are within each partition except for the edges outgoing from what would be considered the “circled” nodes, which must all point into the other partition.
2. Let p_1 denote the cardinal partition and p_2 denote the diagonal partition.
3. All outgoing edges from each node must travel the same “distance” when mapped onto the grid, so that a number can be placed onto the grid square.
4. Each vertex in p_1 must have an equivalent vertex in p_2 (the cardinal and diagonal versions) that has outgoing edges that represent diagonal movement with the proper distance.
5. Circled grid squares must have edges from p_1 travel diagonally into p_2 and edges from p_2 travel cardinally into p_1 .

These conditions are sufficiently strict that generating the state graph without considering the puzzle rules when doing so, followed by attempting to perform a mapping, is nearly guaranteed to fail on every attempt. Thus, a method that involves considering the high-level problem instance and defining the neighbor states in terms of the puzzle and not the state graph itself is the chosen solution.

4.3.3 Search Options

There is a plethora of search algorithms that one can use to select neighbor states, all with some measure of success. We focused on two primary search methods: stochastic local search (with random restart) and simulated annealing.

Stochastic local search (sls) involves iterating over ones neighbors in a random order, and accepting the first neighbor state encountered that has a higher score or rating (according to the objective function) than the current state. If all neighbors result in the

same or a worse state, then the algorithm terminates. We used a fully random algorithm to generate a maze instance, then applied the sls search algorithm, and took the best result. Also, starting the search algorithms from the maze instances provided in Mad Mazes and SuperMazes proved successful as well. Sls often results in being stuck in local maxima/plateaus in the search space because it cannot, at any point in the algorithm, accept a worse state than the current (Russell and Norvig, 2022, 111-114).

Simulated annealing (sa) is similar to sls in that it involves iterating over one's neighbor's in a random order and always accepts higher neighbor states, but it also may accept inferior neighbor states. The probability of acceptance is based on how poor the neighbor state is (a lot worse is less likely to be accepted) combined with a "temperature." The probability of acceptance is given by the following formula $P(A) = e^{\Delta S/T}$ where ΔS represents the difference in value between the neighbor state and the current state and T is the temperature. When ΔS is negative, that is, the neighbor state is worse than the current state, then at high temperatures there will be a high probability of accepting this worse state. However, at low temperatures, it is unlikely that this worse neighbor would be accepted. As $T \rightarrow 0$, the probability of accepting a worse state approaches zero (Russell and Norvig, 2022, 114-115).

Usually, the temperature starts at a maximum value then decreases as the algorithm runs (with each iteration), so near the start of the algorithm it behaves similar to a random walk before starting to climb to higher states. We defined temperature changes after a certain number of iterations. For the first 500 iterations the temperature is 100, then decreases by half every thousand iterations until reaching 5000, at which point it is set to 0.01 (so close to zero that the algorithm behaves like a stochastic local search and only accepts higher or same-rated states).

The temperature is very flexible. It should not be set too high to avoid simulating a fully random walk because that would defeat the purpose of starting from an already excellent maze, but outside of this constraint, as long as the temperature decreases over time, we find good results.

With both sa and sls, we run the search until it terminates; sls when all neighbor states are worse than the current, and sa when the temperature has become zero the algorithm is essentially sls and terminates in the same fashion.

Another search option is to generate *all* neighbor states and accept the one that leads to the highest score, but the overhead in computing all neighbor states is very high and this algorithm was often too slow in practice to yield useful results, especially as puzzle instance size increased.

4.3.4 Neighbor States

We define the neighbor states for each type of maze introduced in Chapter 2 below. These are implemented in Python using generator functions, so we iterate over all possible neighbor states as they are generated one by one; thus, when we choose to accept a state as a successor we have not done the work of generating *all* neighbor states, only those that were evaluated before accepting a successor.

1. Jumping Mazes

For jumping mazes, a neighbor state is simply changing the number and/or the circling of a particular square on the grid. We iterate over the squares in a random order, followed by the numbers and the two circling options. There is an 80% chance that try not circling before circling to help avoid an excess of circled grid squares.

After selecting a square, new number, and circling option, we remove the outgoing edges from each vertex in the state graph that represents this square and store them. Then, we calculate the edges that would result from the new square/number/circling option and add them to the state graph. Running the objective function scores the new state graph and maze instance. Depending on the search function being used, we may accept or reject this neighbor state. If it is accepted, we move to the next iteration immediately. If it is rejected, we remove the new edges added to the state graph, re-add the old edges, and reset the grid square back to its original number and circling.

2. Arrow Mazes

Arrow maze generation is nearly identical to jumping mazes. A random square, arrow direction, color (if present), and circling option are selected (i.e. iterated over in a random order within the neighbor generator function). The state graph and maze are updated with the change, the new instance scored with the objective function, and if accepted we move to the next iteration, otherwise all changes are reset before moving onto the next neighbor.

3. Connections Mazes

Connections maze generation is more complex. In order for the mazes to viewable and solvable by human solvers and not a bewildering jumble of lines, a grid structure is enforced in the maze instance (not the state graph) so that transit lines are only between adjacent vertices in this grid structure (up to 8 adjacencies for each location).

Because the state graph is defined by the transit lines and not the villages themselves, we select a random transit line, color, and line type. Then we choose which village to keep the transit line connected to (one of the two villages to which it is currently connected), and randomly choose a new successor village (from the grid structure). Note that two villages can only be connected by at most one transit line. We store the original characteristics and make the modifications to the state graph and puzzle instance before using the objective function to obtain a new score. This score is passed to the search function which determines, using the previous score of the original instance and the search type, if we accept or reject the new state. Accepting leads to another iteration of search, rejecting leads to resetting all changes before generating the next neighbor.

4. Multiplayer/Spacewreck/Meteor Storm Mazes

Similar to connections mazes, a grid structure is enforced in the puzzle instance so that the generated maze is clean and easily viewable by humans. We randomly choose a corridor and a corridor color, then randomly choose which of the two rooms to keep this corridor connected to (call this room R), randomly choose a new room

from the adjacencies of R that this corridor will now connect to. We also choose new colors for these two rooms. Make the changes in the puzzle instance and state graph, get a new score from the objective function, determine if we accept or reject, reset for next neighbor or move onto the next iteration, etc. as mentioned in the other three maze types.

We discard neighbor states that result in the same current puzzle instance and state graph.

4.4 Example Search Function

In this section, we will provide an example search function used for generation of jumping maze instances with diagonal state changes. Let n denote the number of vertices in the state graph and g denote the number of grid locations. It is best (if possible) to try and scale terms using n to provide weightings that represent importance. Note that we will be attempting to maximize the score of the maze (and not minimize).

The score is initialized to zero. We define the following algorithm to score the state graph and higher-level problem instance.

State graph terms:

1. If there does not exist a path from the start to the finish, then subtract n^3 from the score.
2. Otherwise, count the number of shortest paths from the start to the finish. If there is only one, add $10 * n$ to the score. If there are multiple, do not add $10 * n$.
3. If the length of the shortest path(s) is greater than $n * 0.35$ or less than 11, subtract n^2 from the score. This ensures that the path does not consume too much of the state graph so we can use other parts of it for false paths, and also that we do not have a trivial solution path less than 11.
4. For the following characteristics, they must be calculated for both the state graph and its transpose to ensure adequate difficulty when moving forwards from the start and backwards from the finish. Thus, calculate both cases and take the minimum of the two, and add that to the score.

5. Next we consider the branching and reachability scores. Using a BFS, calculate all descendant vertices and categorize them by distance from the starting vertex. Then, calculate the number of vertices cr that are reachable after a certain number of moves, (10-20% of the shortest path length is a good starting point). Add cr to the score (after calculating cr_t and taking the minimum of cr and cr_t , where cr_t is the same statistic but calculated on the transpose with the finish as the starting point).

For reachability, count the number of vertices r reachable from the start. As mentioned previously, we calculate the proportion reachable from the finish in the transpose graph. Call this r_t . Add the minimum of r and r_t to the score. Then, union the results, which gives all vertices that are reachable and reaching. Add the number of vertices that are either reachable or reaching to the score. Determine all vertices that are in the state graph that are neither reachable nor reaching. Subtract this quantity squared from the score - these represent wasted vertices (isolates).

6. The most important factor in determining the difficulty of a maze, given all other traits are equal, is the decision score that determines exactly what decisions every maze solver will have to make. Compute the required nodes that every path from the start to the finish must go through using the dominance algorithm discussed in Chapter 3 and given in Cooper et al. (2006). Then, we evaluate the decisions at these vertices in both the state graph and its transpose, as discussed in Chapter 3, take the minimum of the two, multiply by two, and add to the score.
7. Now we consider dead ends. A dead end is a reaching or reachable vertex in a state graph that does not have any outgoing edges, (or equivalently in the transpose, does not have any incoming edges). Holes make better traps than dead ends, as mentioned in Chapter 3. Subtract the number of dead ends times five from the score.

4.5 Results

Although we use the objective function defined in the previous section to rate the state graphs, we still need to consider the higher-level maze instances when generating, and these slight additions onto the objective function are defined below for each different type

of mad maze, followed by the generation results.

For each of the mad maze types described below, the size of the puzzle instance was chosen so the state graph has ≈ 128 vertices to enable comparison across the different mazes. This means an 8×8 grid for the grid mazes (jumping and arrow), a connections maze with ≈ 63 transit lines (this will have $63 * 2 = 126$ vertices in the state graph, plus two for the start and end vertices). There will be 32 villages to maintain a similar transit line : village ratio as in Grandpa's Transit Map, which has 36 villages and 70 transit lines. The Spacewreck instances will have 16 rooms, which equates to 121 vertices. Seventeen rooms would result in 137 vertices.

4.5.1 Jumping Maze - Jumping Jim's Encore

The instance objective function terms are defined as follows:

1. It is undesirable to have large areas of the grid that share the same number (Neller et al., 2011, 194-195). Initialize adj-num and adj-circle to zero. Iterate over the grid and calculate the following: for every grid square, evaluate its (eight) neighbors. For each neighbor that shares the same number, add one to adj-num. If the current node is circled and encounters a neighbor that is also circled, add one to adj-circle. Count the number of circled grid squares as well.
2. Subtract adj-num and adj-circle² from the score. Calculate the desirable number of grid squares to be circled (we used 20%). Subtract the difference between the desired number of circled locations and the number of circled locations cubed from the score.
3. Now we consider doubly visiting grid squares, u-turns, and state changes (from cardinal to diagonal to cardinal, etc.). These traits are defined in Chapter 3. We calculate the number of squares that are visited in both the cardinal and diagonal states, the number of u-turns along the shortest path, and the number of state changes between diagonal and cardinal. Add the number of state changes times five and the sum of u-turns plus doubly visited squares times 10 to the score.
4. Subtract n^2 from the score if the starting vertex is circled. If this is allowed, remove this term.

This function yields excellent jumping puzzle instances and state graphs. Table 4.1 shows the results of several mazes generated using this objective function and simulated annealing with random restart.

Table 4.1: Jumping generation maze results are shown below with the primary components of the scores highlighted in **bold**. SP = shortest path, BH = black hole, WH = white hole, R or R = Reachable or Reaching, R and R = Reachable and Reaching. The hole entrances are only to the largest hole. The required decisions is the sum of the required vertices out degrees, which represents the minimum quantity of choices maze solvers will consider. Fwd and bwd decisions are the forward and backward decisions, and the other metrics are previously specified in this chapter and Chapter Two.

Metric/Maze	1	2	3	4	5	6	7	Avg.
SP Length	36	38	34	27	22	36	43	33.7
SP Quantity	1	1	1	1	1	1	1	1
SP Score	1270							
Branching Score	24	31	31	15	15	15	17	21.1
Reachable (%)	68.5	66.9	72.4	72.4	69.3	88.2	91.3	75.6
Reaching (%)	62.2	66.1	74.0	89.0	92.9	89.8	90.6	80.7
R or R (%)	98.4	99.2	99.2	100	98.4	99.2	100	99.2
R and R (%)	32.3	33.9	47.2	61.4	63.8	78.7	81.9	57.0
Reachability Score	199	208	216	218	208	236	241	218
Required Vertices	36	38	33	26	20	32	39	32.0
Required Decisions	82	87	78	60	50	70	84	73.0
Largest BH	35	30	22	8	2	2	2	14.4
BH Entrances	18	16	15	2	1	1	1	7.7
Largest WH	25	24	22	4	17	2	2	13.7
WH Entrances	25	23	14	10	18	6	1	13.9
Fwd Decisions	1024	902	834	832	794	558	418	766
Bwd Decisions	1046	990	850	834	794	560	418	784.6
Decisions Score	1024	902	834	832	794	558	418	766.0
Dead End Score	-50	-50	-60	-50	-80	-60	-60	-58.6
State Graph Score	2467	2361	2291	2285	2207	2019	1886	2216.6
Circled (%)	23.4	21.9	25	21.9	23.4	21.9	23.4	23.0
State Changes	7	7	10	4	6	12	12	8.3
Double Visited	8	10	6	4	3	8	11	7.1
U-turns	9	9	10	9	5	15	17	10.6
Instance Score	143	186	137	99	54	235	276	161.4
Overall Score	2610	2547	2428	2384	2261	2254	2162	2378

Table 4.1 offers several items to consider. The path score (SP score) is the same for every maze. This is due to the $10 * n$ term that results from having a single shortest path,

which has resulted in every maze instance to having a single shortest path. We purposefully included the $10 * n$ to enforce this condition, and we appear to have been successful. In effect, we can consider the base score for all of these mazes to be 1270, which is $10 * n$.

The branching score gives a small bonus onto the score as desired to help prevent long sections of forced moves. If one was generating instances where a large branching factor is more important, then one could add a constant to this term, square it, or multiply it by n , depending on its relative importance.

There are less reachable/reaching vertices as a result of holes in the highest rated mazes. This is a common theme we have noticed when generating these mazes: one can maximize reachability at the cost of traps, which results in a higher reachability score at the cost of a lower decisions score, or one can maximize hole sizes, which results in a higher decision score at the cost of a lower reachability score. In this objective function, we are more heavily weighting the decisions score (it has a constant factor of two applied to it and the way we are calculating it generally leads to higher numbers than the reachability score), hence, the best mazes sacrifice reachability for traps and a higher decisions score. Compare maze one to maze seven. Maze seven has the highest reachability score (241) of the instances presented in Table 4.1 with the highest proportion of reachable (91.3%), reaching (90.6%), and both reachable and reaching (81.9%) vertices. However, it has the lowest decision score.

The decision score dominates the overall score, usually representing about 40%. We claimed previously that the decisions are the most important factor when considering the difficulty of a state graph, and that idea is represented in the scoring metrics that we have chosen.

The highest rated mazes generally have a combination of the most required decisions, (which may or may not be the most required vertices), large black and white holes, and many entrances to these traps from required vertices (which is the BH/WH entrances table entry). Both of the top two mazes exhibit these traits, and both have the highest decision scores. A very low R and R score is indicative of mazes with large unreachability/unreachable

traps; in the most optimal version of these mazes (according to this objective function), only the shortest path vertices are reachable and reaching, and all other vertices are part of large strongly connected holes.

However, large holes are not required to have a large decision score, as maze four shows. Maze four combines several small white holes (of sizes three, three, and four) with one medium size black hole and a large whirlpool such that suboptimal when suboptimal forward decisions do not lead into the black hole, they lead into a whirlpool that costs the solver a lot of time while suboptimal backward decisions lead into false paths and unreachable traps. The higher R and R score indicates the presence of a large portion of vertices that are reachable and reaching, that is, a whirlpool in this case.

We can see the constraint of subtracting (squared) the difference between the desired (20%) and actual number of circled grid squares has been very successful in constraining the instances to $\pm 5\%$ of the desired value. Adding a bonus for state changes, double visited squares, and u-turns has lead to their prevalence in the generated instances, with averages of 8.3, 7.1, and 10.6 respectively. The additional bonus for doubly visiting the start square was also successful with nearly every instance shown doubly visiting the starting square. Adjacent and circled overlap (not shown in the table) have been sufficiently avoided as well.

As designed, the state graph score plays a much larger role than the instance score. When using the same objective function to generate multiple different mazes, it would defeat the purpose of sharing a state graph objective function if it did not play a more significant role than the individual maze instance terms.

The most exceptional of these mazes is depicted in Fig. 4.2 and its component graph is shown in Fig. 4.3. The state graph has the following statistics:

- $n = 127$ vertices.
- One shortest path of length $36/127 = 29\%$ of the state graph.
- After five moves from the start//finish, one can be in $11//7$ distinct locations not considering repeats.
- $87/127 = 68\%$ of vertices are reachable from the start.

6	1	2	(4)	(4)	2	7	2
1	(3)	2	2	(1)	6	(1)	5
2	5	4	4	1	2	(5)	(6)
2	6	1	2	3	2	1	3
3	2	(2)	(4)	4	4	3	2
4	(4)	2	4	2	(4)	2	2
(1)	(6)	4	2	3	(4)	3	4
6	2	(2)	7	5	1	3	GOAL

Figure 4.2: Exceptional jumping maze instance.

- $79/127 = 62\%$ of vertices are reaching (reachable moving backwards from the finish).
- $125/127 = 98\%$ of vertices are reachable or reaching, so there is only two isolates.
- $41/127 = 32\%$ of vertices are reachable and reaching: there are only five vertices that are not on the shortest path that can be reached in both movement directions, all other vertices are part of holes. This indicates near-maximal hole sizes.
- Every vertex on the shortest path dominates the goal. Thus, shortest path vertices and required vertices are synonymous for this puzzle instance, maximizing the possible decision score.
- The largest black hole is 35 vertices with 18 total entrances.
- The largest white hole is 25 vertices with 25 total entrances.

This state graph is exceptional according to the metrics we have defined. It has substantial branching, an optimal number of required vertices, massive black and white hole traps and false paths with many entrances, near-perfect reachability given the large

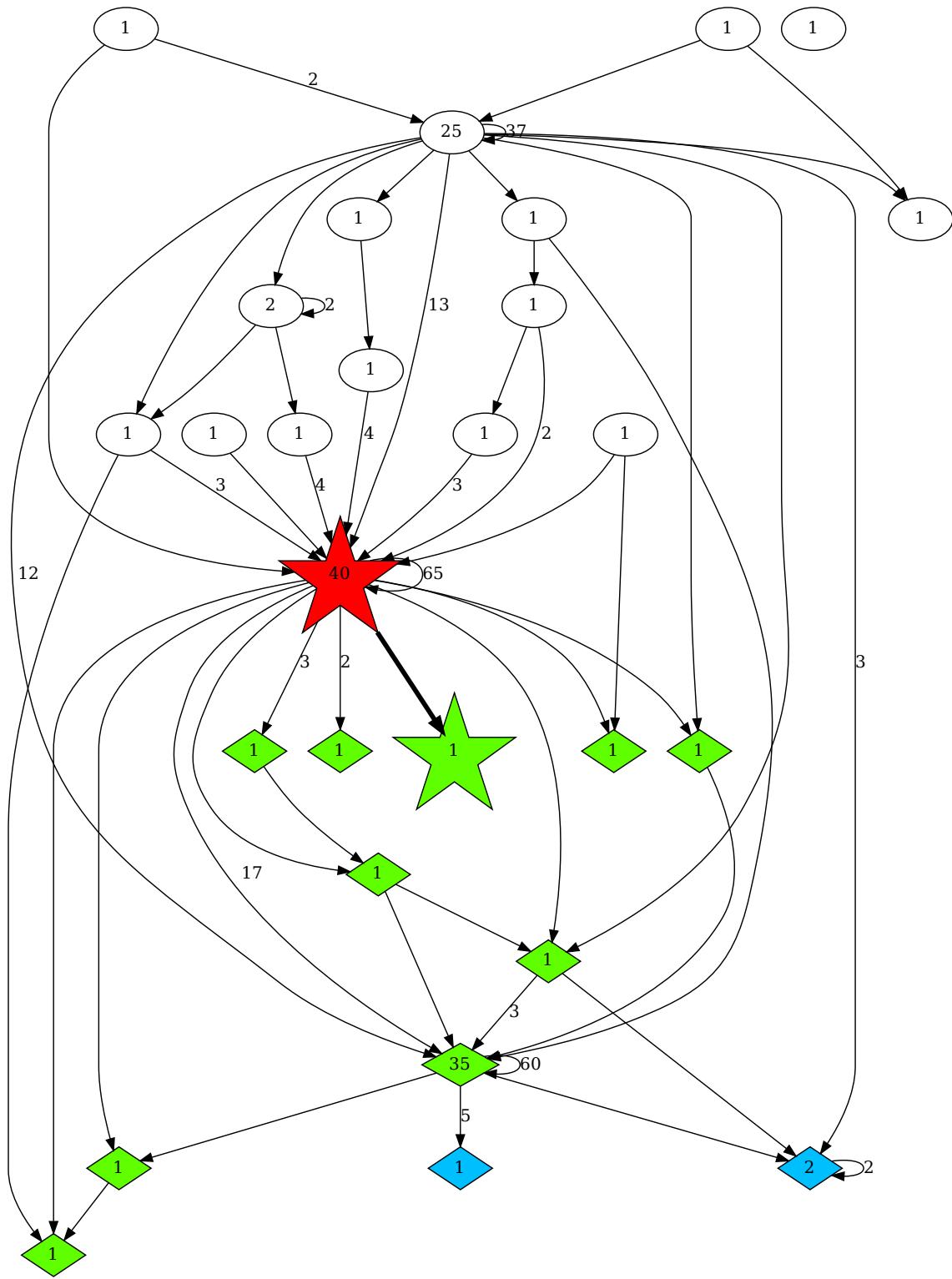


Figure 4.3: Component graph of the excellent jumping maze instance.

traps, and a near-optimal shortest path length. The higher-level maze instance is excellent as well, with the following statistics:

- 15/64 circled grid squares = 23%.
- There are seven swaps between diagonal and cardinal movement, eight grid squares visited in both diagonal and cardinal states, and nine u-turns.

Almost exactly 20% of the grid squares are circled, which was the percentage chosen when generating this maze in the objective function, and there are few areas with large groups of the same number or circling (compare to Fig. 3.1). The shortest path requires seven state changes on a path of length 36, averaging one every five moves. Eight grid squares are visited both diagonally and cardinally, including the starting square (1, 1), and there are nine u-turns along the shortest path, both of which are outstanding for a path of length 36.

This instance is exceptional and it is unrealistic to expect such outcomes on every generation. The problem is over constrained due to the many conflicting traits, which makes optimization particularly difficult. However, the instances presented in Table 4.1 plus 30 additional instances with an average score of about 2250 were generated within an hour, thus demonstrating this method’s consistency.

4.5.2 Arrow Maze - Apollo’s Revenge

The objective function is identical to the jumping maze, including the puzzle instance terms. Because both of these mazes are played on a grid, the puzzle instance metrics optimized in the jumping mazes have a near-identical definition once we consider arrow directions instead of numbers. Circling percentage, state changes, doubling back, and u-turns are identical, and we want to avoid large groups of arrows pointing the same direction and large groups of circled grid squares.

The generation results are shown in Table 4.2. Similar to jumping mazes, we generated many instances and tabulated the statistics for seven, all within the top 25%.

Arrow mazes are similar to jumping mazes in most traits, but they generally have lower scores (2058 average vs 2378 average). The ability to pick any descendant the current arrow points at significantly limits the ability to create traps without an escape. From the

Table 4.2: Apollo’s Revenge generation maze results are shown below with the primary components of the scores highlighted in **bold**. SP = shortest path, BH = black hole, WH = white hole, R or R = Reachable or Reaching, R and R = Reachable and Reaching.

Metric/Maze	1	2	3	4	5	6	7	Avg.
SP Length	30	31	31	35	37	32	31	32.4
SP Quantity	1	1	1	1	1	1	1	1
SP Score	1270							
Branching Score	30	32	37	15	26	31	36	29.6
Reachable (%)	70.1	69.3	70.1	94.5	78.0	90.6	67.7	77.2
Reaching (%)	59.8	63.0	61.4	79.5	77.2	76.4	59.8	68.2
R or R (%)	95.3	98.4	96.9	99.2	96.9	97.6	96.9	97.3
R and R (%)	34.6	33.9	34.6	74.8	58.3	69.3	30.7	48.0
Reachability Score	160	200	184	225	204	211	182	195.1
Required Vertices	30	31	31	35	36	28	31	31.7
Required Decisions	83	83	88	95	89	68	82	84.0
Largest BH	14	14	14	1	1	1	5	7.1
BH Entrances	14	18	23	0	0	0	15	10.0
Largest WH	24	15	25	1	12	1	19	13.9
WH Entrances	20	24	20	0	13	0	18	13.6
Fwd Decisions	836	718	708	488	494	456	490	598.6
Bwd Decisions	840	734	766	494	494	440	578	620.9
Decisions Score	836	718	708	488	494	440	490	596.3
Dead End Score	-160	-115	-130	-140	-130	-145	-130	-135.7
State Graph Score	2136	2105	2069	1858	1864	1807	1848	1955.3
Circled (%)	21.9	23.4	21.9	21.9	20.3	23.4	23.4	22.3
State Changes	10	9	14	13	12	13	14	12.1
Double Visited	3	2	4	5	7	4	4	4.1
U-turns	10	6	4	9	9	9	4	7.3
Instance Score	105	65	79	146	138	117	66	102.3
Overall Score	2241	2170	2148	2004	2002	1924	1914	2057.6

statistics in Table 4.2, one can see that only the best of the mazes generated have nontrivial black and white holes, which can substantially increase the decision score. Hence, lower decision scores on average when compared to jumping mazes, even though there is on average a higher number (about 10 more on average) of required decisions—these decisions are simply not worth as much because most of them do not lead into large traps.

Due to the forward and backwards movement, it is impossible to move out of both vertices in the state graph that represent the corners. It is similarly difficult for other squares on the edges of the grid unless they point directly North or South. This leads to a

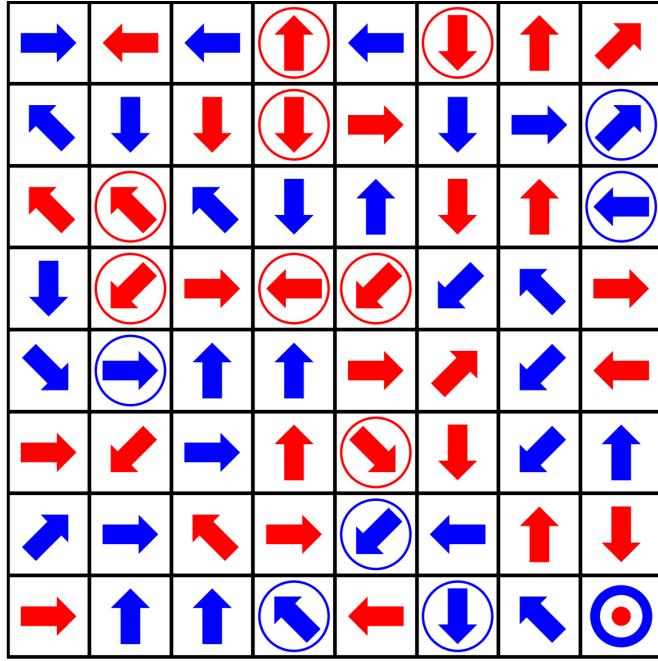


Figure 4.4: Best Apollo’s Revenge maze instance.

much larger dead ends penalty when compared to jumping mazes (-135.7 average vs -58.6 average). The best instance generated follows the paradigm of large holes with many entrances, and thus a poor R and R score. However, it has much fewer holes and more dead ends when compared to the exceptional jumping maze instance, which follows from the properties of the maze.

4.5.3 Connections Maze - Grandpa’s Transit Map

Three terms are added to the objective function to account for the higher level traits of connections mazes. Firstly, we add $10 * n$ to the score if there are two or less shortest paths. This enables us to generate the interesting mazes as mentioned previously that trace a long path out into the maze, turn around, and follow the same transit lines in reverse order. There will always be two possible ways to take this loop, so we allow for two shortest paths. We count the number of times we visit the same village multiple times and when we use a transit line in both directions (these account for doubling back in

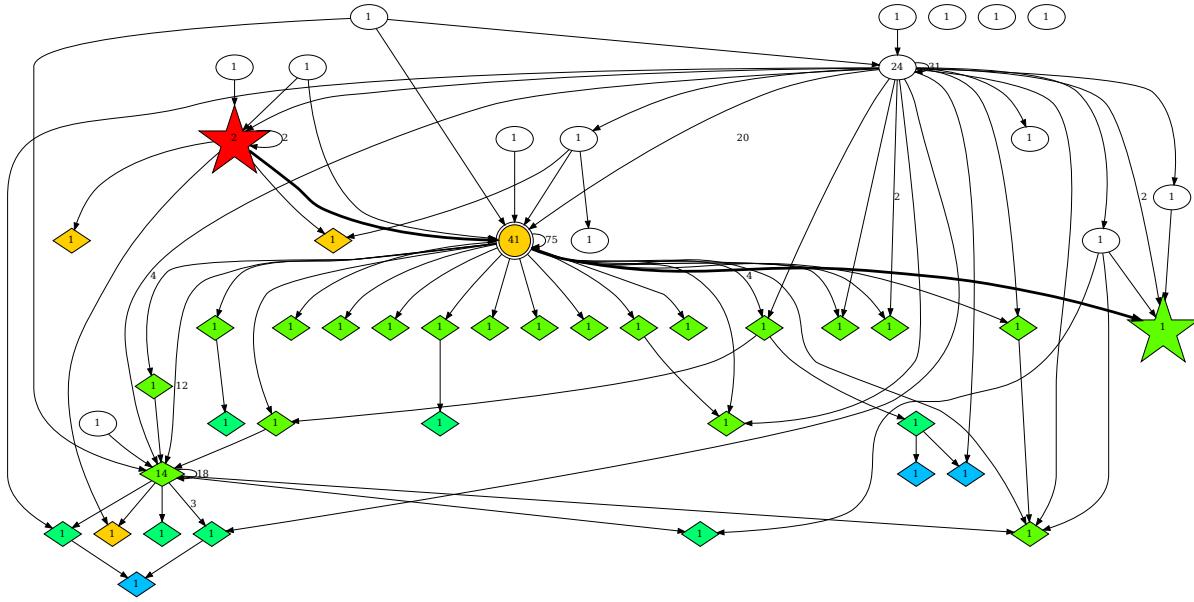


Figure 4.5: Component graph of the best Apollo’s Revenge instance. Compared to the jumping maze instance in Fig. 4.3, it has many more dead ends and smaller holes.

connections mazes). We add three times the number of same village visits and 10 times the number of double line visits to the score.

Lastly, we want to ensure there is an equal distribution of village line types and colors. Count the number of each type/color, calculate the difference between the most prevalent and least prevalent line type, multiply this difference by the number of different line types, and finally subtract this number from the score. Repeat the process for the line colors.

A primary feature of connections mazes is connectivity and reachability. It is often the case that nearly 100 percent of the vertices are both reachable and reaching, leading to the maximum possible reachability score. This can conflict with the potential decision score that would result from large traps, however. When generating connections maze instances with this objective function, there is a large group of local maxima that result from maximizing the reachability score, and it can be difficult for the search to instead find an instance with large traps, which leads to a much larger decision score and overall score.

After generating many mazes using simulated annealing and random restart, we found

several mazes with large traps (about 1 in 10 had large traps).

Another possible method is to run simulated annealing with random restart until it creates a maze with some traps, and then running simulated annealing on that instance with a restricted temperature (say, only 5-10), encouraging only small steps down to avoid the large local maxima.

Table 4.3: Grandpa’s Transit generation maze results are shown below. There are 128 vertices in each state graph, and Two SPs Addback is the $10 * n$ given from having two shortest paths. When a line/village is visited multiple times, this is counted in Village/Line Doubling. The color traits penalize the score for unequal color distributions.

Metric/Maze	1	2	3	4	5	6	7	Avg
SP Length	39	40	38	38	29	30	35	35.6
SP Quantity	2	2	2	2	2	2	2	2
SP Score	0							
Branching Score	26	35	28	31	7	10	30	23.9
Reachable (%)	69.5	70.3	68.0	68.8	88.3	96.9	64.1	75.1
Reaching (%)	69.5	70.3	69.5	70.3	88.3	95.3	69.5	76.1
R or R (%)	100	100	100	100	100	100	100	100
R and R (%)	39.1	40.6	37.5	39.1	76.6	92.2	33.6	51.2
Reachability Score	216	217	214	215	240	249	209	222.9
Required Vertices	36	37	35	35	26	27	32	32.6
Required Decisions	92	91	90	88	51	53	71	76.6
Largest BH	29	29	29	28	4	3	30	21.7
BH Entrances	35	32	36	31	5	1	28	24
Largest WH	29	29	29	28	4	1	30	21.4
WH Entrances	35	31	37	32	5	2	29	24.4
Fwd Decisions	1922	1776	1768	1522	1362	1148	1228	1532.3
Bwd Decisions	1916	1760	1814	1602	1546	1148	1166	1564.6
Decisions Score	1916	1760	1768	1522	1362	1148	1166	1520.3
Dead End Score	-25	-25	-15	-20	-15	-20	-35	-22.1
State Graph Score	2133	1987	1995	1748	1594	1387	1370	1744.9
Village Doubling	21	21	19	19	12	12	14	16.9
Line Doubling	14	14	13	13	9	8	10	11.6
Village Colors	-32	-12	-12	-4	-8	-4	-36	-15.4
Line Colors	-15	0	-6	-6	0	0	-9	-5.1
Instance Score	156	191	169	177	118	112	97	145.7
SP Addback	1280							
Overall Score	3569	3458	3444	3205	2992	2779	2747	3170.6

The results for this objective function are given in Table 4.3. All of the mazes have two shortest paths and line doubling, which means they all have some form of loop that can be

traced in multiple directions as discussed previously.

The reachability score average is 223, higher than jumping and arrow mazes, indeed every maze has zero isolates and this score is below 210 for only one instance. The bidirectionality inherit within transit lines enables better reachability, which is why we see this general increase in both the reachability statistics and scores. Even the mazes with large holes have better reachability scores and statistics when compared to the jumping and arrow mazes with similar characteristics. Similarly, the dead ends penalty is much lower on average (-22.1 compared to -135.7 and -58) for the same reasons.

We see the same two paradigms of this objective function emerge in the connections maze results as in the grid mazes. The highest-scoring mazes (top four) have large holes with many entrances and required decisions that lead into the holes. The average to medium scoring mazes (five to six) exhibit excellent reachability and few unreaching traps although some have whirlpools.

The highest-scoring maze generated by this algorithm is nearly optimal when considering its characteristics. Considering the state graph in Fig. 4.7, there is sufficient branching to avoid forced moves at the start (we can see that one edge leads directly into the black hole, so there is a 50% chance of a solver immediately entering it), two shortest paths (so it gets the $10 * n$ SP Addback), zero isolates, and still a substantial reachability score (216). The largest contributor to the score is the decision score, and we can see why this is the case from the state graph. Of the length 39 shortest path, 36 are required vertices. At these 36 vertices, every incorrect decision leads either into the black hole or requires many steps to be retraced in the whirlpool, and when retracing these steps one can easily end up in the black hole as well.

The decision score is not only the exceptional trait: this maze also exhibits the tricky property that I mentioned previously: the shortest path retraces 14 of the same lines in both directions. It traces out a path into the maze, turns around, and then traces the same lines in the reverse order before turning towards the solution.

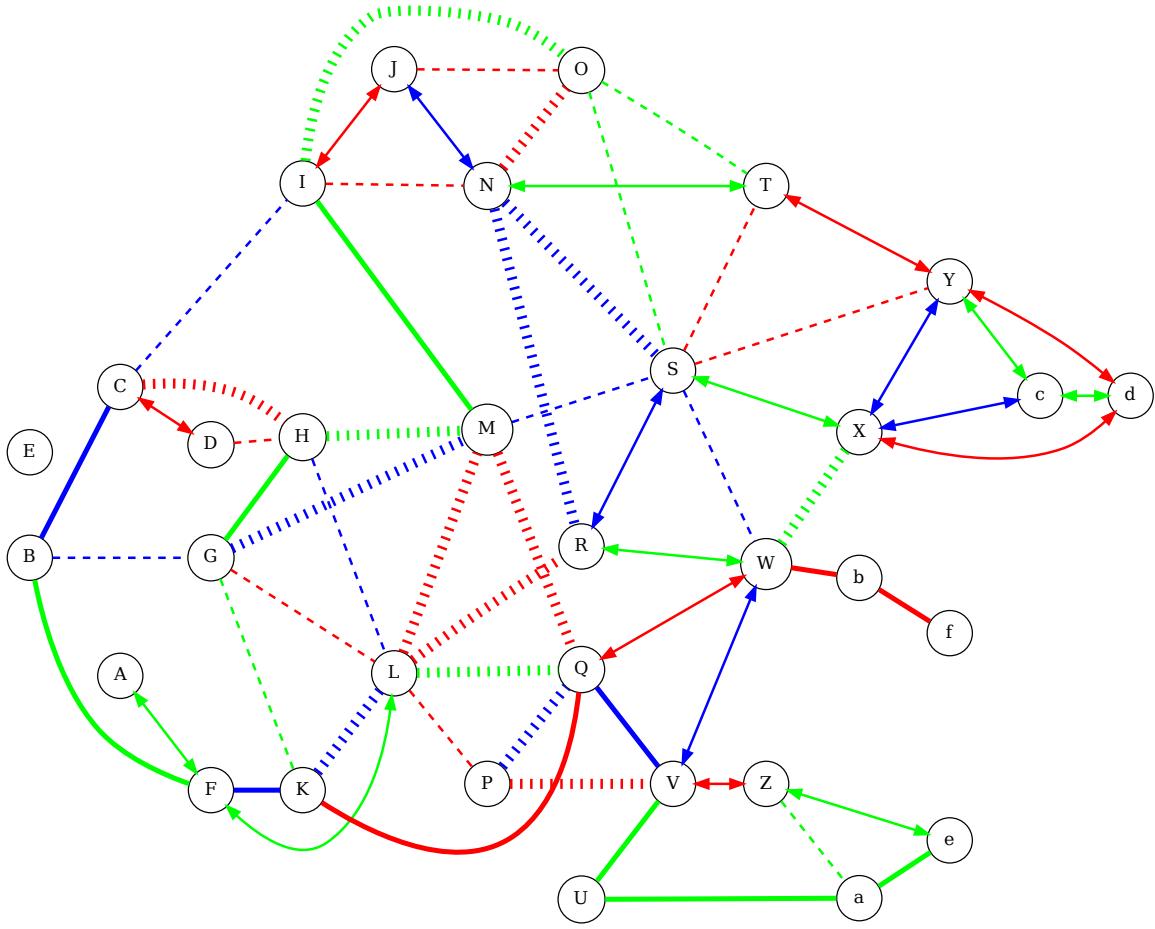


Figure 4.6: Granpa’s Transit Map instance one. Note that E is an isolated village without any transit lines. This means we could potentially increase the number of transit lines when generating these instances. One of the two shortest paths is A F L Q M L R N S R W X S O T S Y d c X d Y S T O S X W R S N R L M Q W b f.

4.5.4 Multiplayer Maze - Spacewreck

The additional objective function terms for Spacewreck are similar to Grandpa’s Transit Map. We start by counting the number of rooms and corridors of each color. However, unlike in the previous maze, we must strictly enforce that there is an even distribution among the colors or the search will result in 90% of the rooms and corridors being the same color. One can understand intuitively based on the rules of this maze that a nearly monochromatic maze will have the highest reachability and the fewest isolates.

However, this often leads to long sequences of forced (or obvious) moves. Consider the case where both parties are in rooms of the most prevalent color (which happens often), and the outgoing corridor(s) from p1 are not the most prevalent color. P2 must find a path to a room of one of the less prevalent colors following edges of the most prevalent color, which leads to long sequences of forced and obvious moves.

However, enforcing the even color distribution too harshly can result in a very large number of isolates, which is wasted space in the state graph as mentioned previously. Thus, calculate the difference between the least prevalent color and the most prevalent color for both rooms and corridors separately. Subtract each of these values **cubed** from the score to enforce equal coloring. Cubing this term instead of multiply by n before subtracting allows there to be small differences in color distribution without a large penalty, but the penalty grows rapidly as the difference increases. This results in mazes with color difference between the min and max of 1-4, which is ideal.

Similar to doubling back in previous mazes, we count the number of times that a repeated room appears in the shortest path and add this quantity times three to the score. A tricky aspect of this maze is it is possible for both players to be in the same room, which may seem counterintuitive at first to human solvers. We count the number of times both players are in the same room and add this number times 20 to the score.

We used the state graph objective function modified with these additional instance terms to generate many Spacewreck instances. The statistics of seven instances are presented in Table 4.4.

The results are similar to the other mazes with the two paradigms (reachability vs holes) although most of the mazes fall into the holes category. Compared to the other maze types, the scores are generally lower, similar to arrow mazes. These mazes are slightly handicapped by having seven fewer vertices in the state graph, which would be an increase of 70 on the SP Score and lead to increases in other categories as well.

The top two mazes are tied. It's interesting that the first maze has a lower decisions score, but makes up the difference with a combination of reachability and branching. Maze

Table 4.4: Spacewreck generation maze results are shown below. Num Colors is the number of colors used in creation of the maze instance (two or three). Room repeating is the count of the same room being visited multiple times, Same Room is the count of the number of times the two players must enter the same room, and the colors are penalties for uneven color distributions.

Metric/Maze	1	2	3	4	5	6	7	Avg.
Num Corridors	42	42	42	42	42	31	42	40.4
Num Colors	2	3	3	2	3	2	2	2.4
SP Length	26	22	27	23	31	31	25	26.4
SP Quantity	1	1	1	1	1	1	1	1
SP Score	1210							
Branching Score	28	13	25	21	24	25	17	21.9
Reachable (%)	56.2	59.5	63.6	71.9	63.6	73.6	86.8	67.9
Reaching (%)	71.1	57.9	62.0	73.6	65.3	81.8	93.4	72.1
R or R (%)	99.2	96.7	97.5	100.0	95.9	99.2	97.5	98.0
R and R (%)	28.1	20.7	28.1	45.5	33.1	56.2	82.6	42.0
Reachability Score	186	170	183	207	167	207	213	190.4
Required Vertices	26	22	27	21	31	31	24	26.0
Required Decisions	78	48	57	61	68	73	68	64.7
Largest BH	16	28	28	21	20	14	3	18.6
BH Entrances	29	20	18	8	21	11	2	15.6
Largest WH	23	27	26	8	23	12	3	17.4
WH Entrances	31	20	25	17	24	11	5	19.0
Fwd Decisions	814	866	854	778	646	546	506	715.7
Bwd Decisions	918	858	886	772	740	618	504	756.6
Decisions Score	814	858	854	772	646	546	504	713.4
Dead End Score	-55	-70	-105	-45	-85	-30	-50	-62.9
State Graph Score	2183	2181	2167	2165	1962	1958	1894	2072.9
Room Doubling	11	10	10	10	13	14	10	11.1
Same Room	2	3	1	1	5	1	2	2.1
Room Colors	-1	-8	0	-27	0	-1	-1	-5.4
Corridor Colors	0	-8	-8	-8	0	-1	0	-3.6
Instance Score	72	74	42	15	139	60	69	67.3
Overall Score	2255	2255	2209	2180	2101	2018	1963	2140.1

three also has a larger decision score than maze one, but maze one makes up this difference through less dead ends and a higher instance score. Maze one demonstrates that even the lower magnitude scoring metrics can significantly impact the score.

Note that there is a difference in the number of corridors and also the number of colors in many of these instances, and yet they all scored well. Two mazes tied for the highest

score of 2,255, one with two colors and one with three colors. Choosing the number of colors allowed for the rooms and corridors was difficult. Abbott uses four colors for 28 rooms and 40 corridors, but his Spacewreck instance has a lot of isolates. Minimizing isolates and having an even color distribution is difficult, especially with only 16 rooms. With only two colors, we were able to get good results with even only 31 corridors (maze six), but with three colors, we could not get a maze over 2000 with only 31 corridors. Thus, we elected to use the largest number of corridors the grid structure underneath would allow, which was 42, given that we were not allowing multiple corridors between two rooms, even if the corridors were in opposite directions with different colors. The results demonstrate that it is possible to have a high scoring maze with three colors, 42 corridors and only 16 rooms.

Due to having two players, small changes to the maze instance (which is what we choose between when generating) can result in large changes to the state graph, which makes local search more difficult for this type of maze. The instance scores are lower as well, and this is due to the color constraints and generally fewer notions of “doubling back” to increase the score.

It seems most natural for this type of maze to generate holes of substantial size; all of the results except for maze seven have substantial holes. Abbott noted in his description of this problem that it has an “abundance of loopiness” (Abbott, 1999, 37). Perhaps the two player color constraints, especially when the board is more monochromatic, enable this behavior.

The topologies of the top mazes are similar to the previous mazes - one large black and white hole combined with a whirlpool of the required vertices (see Fig. 4.9). Additionally, an incorrect choice (one of two) from the starting vertex leads directly into the black hole, similar to Fig. 4.7).

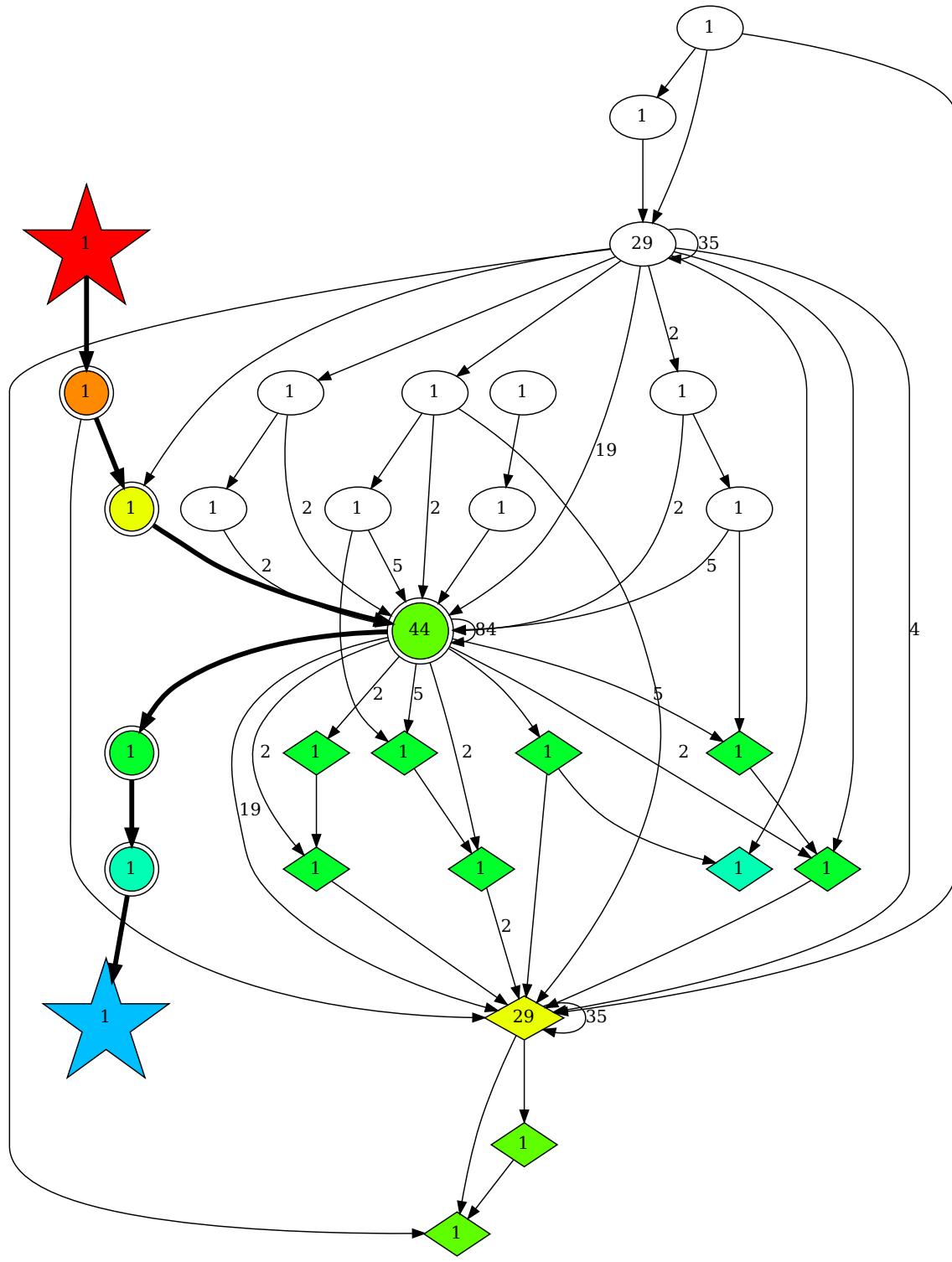


Figure 4.7: Maze one of the connections mazes - exceptional holes and a whirlpool combined with a large number of required decisions.

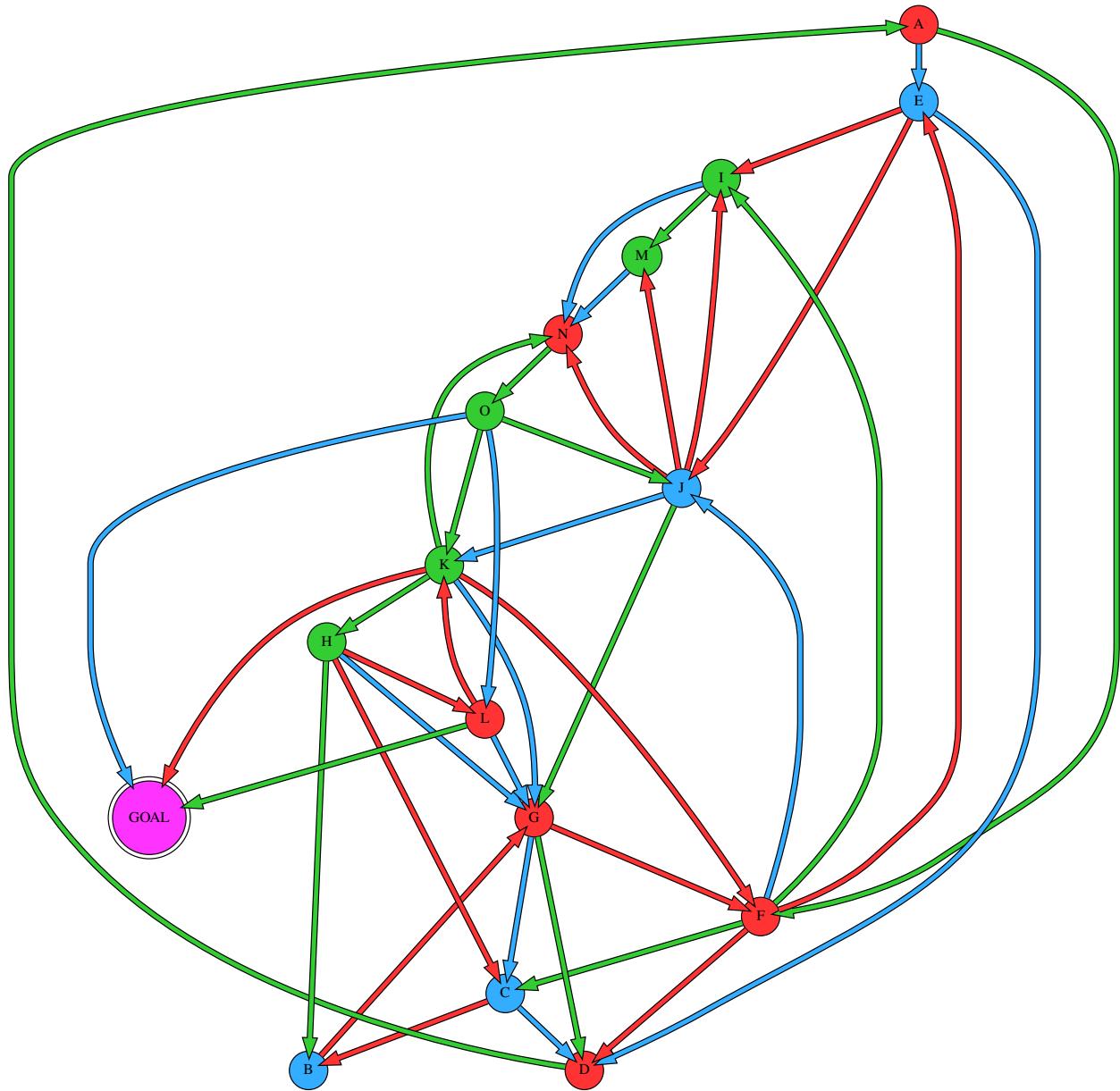


Figure 4.8: The best Spacewreck instance (three color).

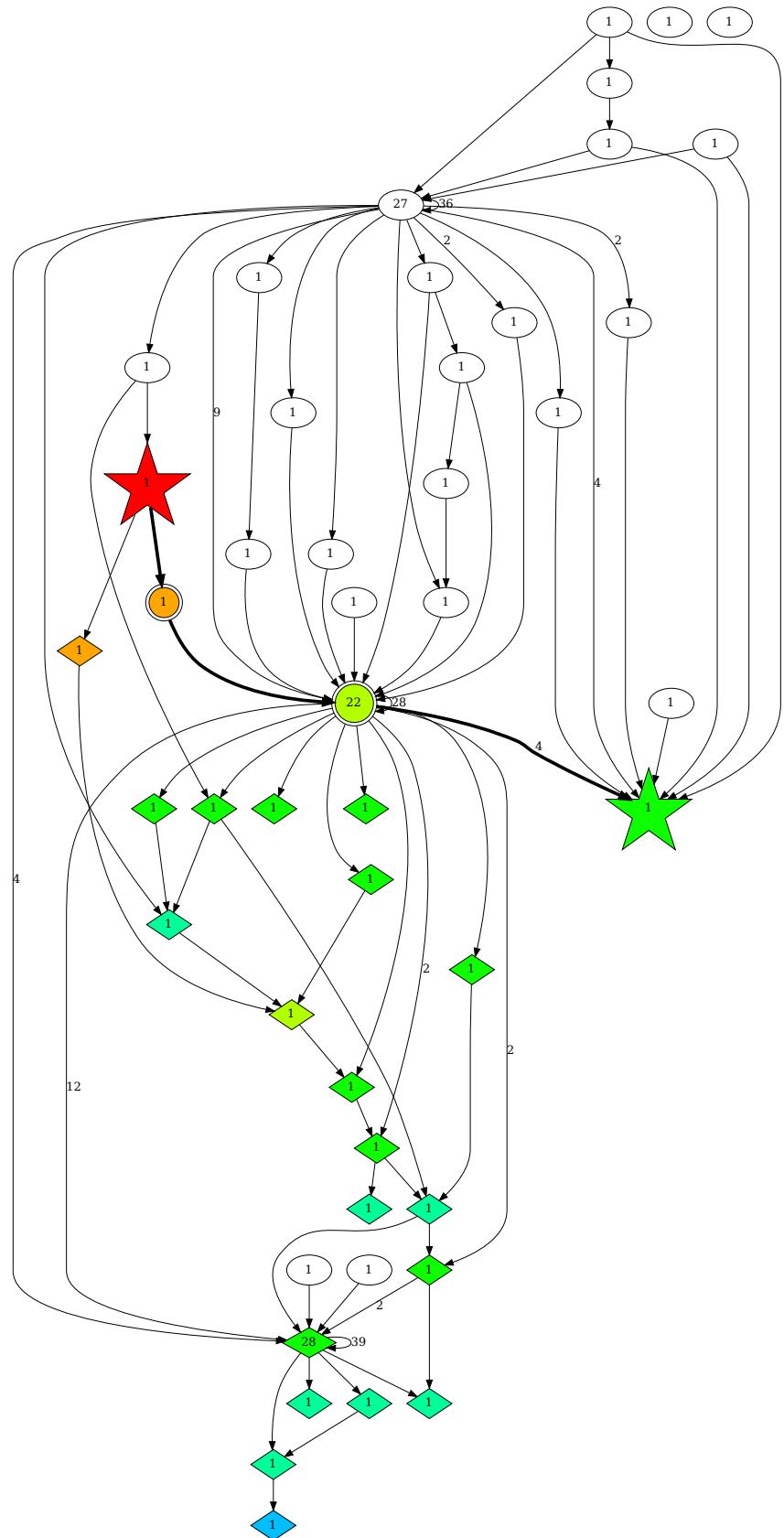


Figure 4.9: The component graph associated with the best Spacewreck instance given in Fig. 4.8.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Future Work

5.1.1 Human Maze Solving

We create mazes to be difficult for *humans* to solve because these are simple to model and solve on a computer. Our approach consisted of creating objective functions and maximizing their value based on maze characteristics. However, these objective functions need to be validated. One possible approach is to simulate how a human would solve these mad mazes and compare the simulation results to the objective function.

There is a plethora of research on human methods to solve classic mazes, but none that we have found on mad mazes. Thus, we have looked into the classic maze methods to find potential similarities to mad maze solving method. Determining human methods to solve mad mazes and extrapolating classical maze human interaction to logic mazes both need further research.

Zhao and Marquez (2013) identify two “stages” of human maze solving: exploration and guidance. In the exploration phase, the solver’s eyes rapidly trace potential paths through the maze. Once a path has been chosen, the path is traced out (this study was performed digitally, so the guidance phase was the subject guiding the mouse along the chosen path). Complex mazes lead to often multiple stages of exploration and guidance (Zhao and Marquez, 2013, 6-8). On smaller mazes, some solvers attempted a brute force method, tracing paths through the maze continually until they found the solution, but this makes mistakes very costly and a more analytical approach was often used on larger mazes.

Karlsson used a DFS to simulate human maze solving, although he mentioned a DFS is not a perfect simulation because human choices at intersections are non-deterministic (Karlsson, 2018, 2). Although we are running this human-simulated algorithm on the state graph, it is as if we implicitly discover the state graph, as a human would not enumerate the entire state graph before solving the maze. We ultimately decided to use a randomized

DFS with many runs and the average number of moves taken before reaching the solution was used as the difficulty metric. The results were somewhat similar to the objective functions that we created, but often were more ordered by the number of vertices in the state graph, which naturally leads to longer average DFS completion times. Additional work to identify a suitable algorithm that sufficiently models human state-space exploration could be used as another difficulty rating for logic mazes.

It is possible for games such as Sudoku to be modeled as constraint satisfaction problems, and Pelánek (2011) has been able to model human Sudoku solving behavior with success. However, comparisons to games such as Sudoku are difficult in this context because they are fundamentally different problems. Sudoku is a constraint satisfaction problem and has been proven to be NP-Complete in the general case. An analysis of Sudoku difficulty presented by Pelánek showed an excellent correlation with human solving of 0.85 (Pelánek, 2011, 2-6), but additional work is necessary to determine if this could be mapped to logic mazes.

5.1.2 Additional Logic Mazes

This work could be continued for several other types of Mad Mazes presented by Abbott, such as Alice/Step Change Mazes, Rolling Cube Mazes, Theseus and the Minotaur, and more. Theseus and the Minotaur has been turned into a game and monetized as a mobile app (Abbott, 2010, 1). One could theoretically create such an app to test the difficulty of multiple different generated logic mazes with human players.

5.1.3 State Space Puzzles

Any game that can be modeled as a state space can be represented using this model, and its characteristics and traits rated by the scoring methods we have developed. This has potential for applications in the gaming industry (a multi-billion dollar industry that continues to grow). Modeling a game as a state space and identifying qualities the space must contain to be entertaining and challenging for humans to solve is a potential application. Generally, the state space should look like a state space of a maze, especially

in puzzle games. Many other games such as sokoban and can be represented as state spaces (Jarusek and Pelánek, 2010, 2) and rated in such a fashion as the logic mazes presented in this work.

5.1.4 Explore other optimization techniques

There is an abundance of literature in the AI based searches; we used a relatively simple search method as a proof-of-concept. Other methods may produce better results with certain state graphs or mazes. Additionally, it may be possible to reduce the maze-generation problem to SAT or ILP, and then use an SMT solver such as Z3 or an ILP solver. This is another potential avenue for future research.

5.1.5 State Graph to Maze Instance

As noted in Chapter Three, it is theoretically possible to build a perfect state graph that has all of the necessary properties of the puzzle instance, and invert it back to the puzzle. Whether it is possible to formally characterize such a state graph in graph theoretic terms is itself an interesting research question. If the answer is yes, the next research task would involve developing algorithms to generate such state graphs.

5.1.6 Applications in pedagogy: Puzzle-based Learning

Puzzle-based learning is a pedagogical strategy which teaches problem-framing and problem-solving for unstructured problems (Meyer et al., 2014, vi). The authors of this work claim these skills are generally lacking in today's students. A possible area of future work would involve a more formal pedagogical exploration of mad mazes as part of a puzzle-based learning strategy that assumes students have background in basic graph algorithms and concepts.

5.2 Conclusions

We have answered the research question of how to intelligently generate mad mazes instances: using state graph representations, objective functions, and local search. It is vital to consider not only the state graph's characteristics, but also the qualities of the maze instance itself to avoid undesirable properties such as monochromatic puzzle

instances that are designed to be poly-chromatic. Both stochastic local search and simulated annealing provided desirable results in the mazes that were generated. The most notable contribution of this work is that the same objective function can be used to rate any mad maze state graph, and potentially even outside mad mazes, to any game that can be modeled as a state graph.

REFERENCES

- Abbott, R. (1990). *Mad Mazes: Intriguing Mind Twisters for Puzzle Buffs, Game Nuts and Other Smart People*. Adams Media Corporation.
- Abbott, R. (1997). *Supermazes: Mind Twisters for Puzzle Buffs, Game Nuts and Other Smart People*. Prima Publishing.
- Abbott, R. (1999, Dec). Logic mazes. <https://www.logicmazes.com/dec99.html>.
- Abbott, R. (2010, Jul). Theseus and that pesky minotaur. <https://www.logicmazes.com/theseus.html>.
- Buchin, K., M. Buchin, E. Demaine, M. Demaine, D. El-Khechen, S. Fekete, C. Knauer, A. Schulz, and P. Taslakian (2007, 01). On rolling cube puzzles. pp. 141–144.
- Cooper, K., T. Harvey, and K. Kennedy (2006, 01). A simple, fast dominance algorithm. *Rice University, CS Technical Report 06-33870*.
- Fisher, A. and G. Gerster (2000). *The Art of the Maze*. Seven Dials.
- Hagberg, A. A., D. A. Schult, and P. J. Swart (2008). Exploring network structure, dynamics, and function using networkx. In G. Varoquaux, T. Vaught, and J. Millman (Eds.), *Proceedings of the 7th Python in Science Conference*, Pasadena, CA USA, pp. 11 – 15.
- Jarusek, P. and R. Pelánek (2010). Human problem solving : Sokoban case study.
- Karlsson, A. (2018). Evaluation of the complexity of procedurally generated maze algorithms.
- Meyer, E. F., N. Falkner, R. Sooriyamurthi, and Z. Michalewicz (2014). *Guide to Teaching Puzzle-based Learning*. Springer London.
- Neller, T. W., A. Fisher, M. T. Choga, S. M. Lalvani, and K. D. McCarty (2011). Rook jumping maze design considerations. In H. J. van den Herik, H. Iida, and A. Plaat (Eds.), *Computers and Games*, Berlin, Heidelberg, pp. 188–198. Springer Berlin Heidelberg.
- Pelánek, R. (2011, 01). Difficulty rating of sudoku puzzles by a computational model.
- Russell, S. J. and P. Norvig (2022). *Artificial Intelligence: A Modern Approach* (4 ed.). Pearson.
- Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer Publishing Company, Incorporated.
- Zhao, M. and A. G. Marquez (2013). Understanding humans' strategies in maze solving. *CoRR abs/1307.5713*.