Evaluation of the Complexity of Procedurally Generated Maze Algorithms

Albin Karlsson
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Digital Game Development. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Albin Karlsson
E-mail: alkf14@student.bth.se

University advisor:
M.Sc Diego Navarro
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden
Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
**Abstract**

**Background.** Procedural Content Generation (PCG) in Video Games can be used as a tool for efficiently producing large varieties of new content using less manpower, making it ideal for smaller teams of developers who wants to compete with games made by larger teams. One particular facet of PCG is the generation of mazes. Designers that want their game to feature mazes also need to know how to evaluate their maze-complexity, in order to know which maze fits the difficulty curve best.

**Objectives.** This project aims to investigate the difference in complexity between the maze generation algorithms recursive backtracker (RecBack), Prim’s algorithm (Prims), and recursive division (RecDiv), in terms completion time, when solved using a depth-first-search (DFS) algorithm. And in order to understand which parameters affect completion time/complexity, investigate possible connections between completion time, and the distribution of branching paths, distribution of corridors, and length of the path traversed by DFS.

**Methods.** The main methodology was an implementation in the form of a C# application, which randomly generated 100 mazes for each algorithm for five different maze grid resolutions (16x16, 32x32, 64x64, 128x128, 256x256). Each one of the generated mazes was solved using a DFS algorithm, whose traversed nodes, solving path, and completion time was recorded. Additionally, branch distribution and corridor distribution data was gathered for each generated maze.

**Results.** The initial results showed that mazes generated by Prims algorithm had the lowest complexity (shortest completion time), the shortest solving path, the lowest amount of traversed nodes, and the lowest proportion of 2-branches, but the highest proportion of all other branch types. Additionally Prims had the highest proportion of 4-6 length paths, but the lowest proportion of 2 and 3 length paths.

Later mazes generated by RecDiv had intermediate complexity, intermediate solving path, intermediate traversed nodes, intermediate proportion of all branch types, and the highest proportion of 2-length paths, but the lowest proportion of 4-6 length paths.

Finally mazes generated by RecBack had opposite statistics from Prims: the highest complexity, the longest solving path, the highest amount of traversed nodes, the highest proportion of 2-branches, but lowest proportion of all other branch types, and the highest proportion of 3-length paths, but the lowest of 2-length paths.

**Conclusions.** Prims algorithm had the lowest complexity, RecDiv intermediate complexity, and RecBack the highest complexity. Increased solving path length, traversed nodes, and increased proportions of 2-branches, seem to correlate with increased complexity. However the corridor distribution results are too small and diverse to identify a pattern affecting completion time. However the corridor distribution results are too diverse to make it possible to discern a pattern affecting completion time by just observing the data.

**Keywords:** PCG, Mazes, Labyrinth, Recursive Backtracker, Recursive Division, Prims Algorithm, Randomized Prims Algorithm, Depth-first-search, Game
Acknowledgments

I would like to thank my supervisor, M. Sc Diego Navarro for his clear, to-the-point, actionable advice. Without him, this thesis would probably be a mess. I would also like to thank my fellow students sitting in the computer lab for being such good company, they helped making the weeks of writing and coding a blast.
# Contents

Abstract i
Acknowledgments iii

1 Introduction 1
  1.1 Motivation .................................. 1
  1.2 Research Questions ............................ 2
  1.3 Thesis Layout .................................. 2

2 Background 5
  2.1 Theoretical Background ........................ 5
     2.1.1 Graph ................................... 5
     2.1.2 Spanning Tree ............................. 6
     2.1.3 Weighted Graph ............................ 6
     2.1.4 Greedy Algorithm .......................... 6
     2.1.5 Procedural Content Generation ................. 6
     2.1.6 Perfect Maze ................................ 7
     2.1.7 Randomized Prim’s Algorithm ................. 7
     2.1.8 Recursive Backtracker / Depth-First-Search Algorithm .... 8
     2.1.9 Recursive Division Algorithm .................. 8
  2.2 Related Work .................................. 9

3 Method 13
  3.1 Development Tools .............................. 13
  3.2 Implementation Details .......................... 14
  3.3 DFS Solver and Completion Time ................. 14
  3.4 Branches .................................... 15
  3.5 Path Length .................................. 16
  3.6 Corridor Distribution ............................ 16
  3.7 Experiment Procedure ............................ 16

4 Results 17
  4.1 Maze Appearance ............................... 17
  4.2 Completion Time ................................ 18
  4.3 Path Length ................................... 21
  4.4 Corridor Distribution ............................ 23
  4.5 Branch Distribution ............................. 25
# Analysis and Discussion

5.1 Completion Time ...................................................... 29
5.2 Branches, Path Length, and Corridors Effect On Completion Time ...................................................... 31
  5.2.1 Branches ...................................................... 33
  5.2.2 Path Length ...................................................... 33
  5.2.3 Corridors ...................................................... 34

# Conclusions and Future Work

6.1 Future Work ...................................................... 36
References ...................................................... 37

# Maze Generation Algorithms

A.1 Prim’s Algorithm ...................................................... 39
A.2 Randomized Prim’s Algorithm ...................................................... 39
A.3 Recursive Backtracker Algorithm ...................................................... 40
A.4 Recursive Division Algorithm ...................................................... 40

# Code Examples

B.1 Recursive Code ...................................................... 41
B.2 Recursive Backtracker Code ...................................................... 43
B.3 Recursive Division Code ...................................................... 45
B.4 Randomized Prim’s Algorithm Code ...................................................... 46
B.5 Depth-first-search Algorithm Code ...................................................... 49
Computer games are one of the fastest growing mediums today, with a large amount of players spending many hours immersed in digital worlds filled with novel content. So far handcrafted content has been able to keep up with the rising production costs and expectations of players, but if costs and expectations keeps rising it may become unsustainable\(^1\). Procedural Content Generation (PCG) holds the potential to fix this. PCG can aid developers with creating more novel content, faster, while saving man-hours. It typically does this by either automatically generating content to directly be used by the player, or as a creative aid for developers who ultimately chooses the generated content to be used. The challenge with PCG is defining the parameters by which content is generated that satisfies both developers and players(Shaker, Togelius, & Nelson, 2016). One specific usage of PCG is the generation of mazes. Mazes are useful for both level design and game design as a piece of content that can simultaneously be used as a "physical" element of a level, and a puzzle for the player. They are well-suited for PCG since the end result is relatively easy to express algorithmically: a start point, an end point, and a series of puzzling sprawling interconnected paths between them. Some examples of games featuring mazes are the classic Pac-man, and roguelikes such as Hauberk\(^2\), a game developed by Bob Nystrom\(^3\), the author of Game Programming Patterns\(^4\).

1.1 Motivation

Game Designers want to craft a good difficulty curve for their players. If they want mazes in their game, and if it turns out that different maze algorithms generate mazes of different complexity, the designer would want to know which maze algorithm fits the difficulty curve best. When a designer is short on time, like when producing prototypes, it is useful to know the complexity level of mazes to quickly be able to pick a suitably complex maze algorithm.

Previous research made by Kim et. al (Kim & Crawfis, 2015) proposed a framework for a maze generation tool that could perhaps be improved by incorporating the

\(^1\)Steve Thodore, Why have video game budgets skyrocketed in recent years?, https://www.quora.com/Why-have-video-game-budgets-skyrocketed-in-recent-years (accessed 11-04-2018)


Chapter 1. Introduction

results of an investigation of parameters affecting completion time. This improvement could in turn aid developers in generating custom mazes.

Therefore this thesis aims to evaluate the complexity of different maze generation algorithms in terms of maze-completion time, and to understand what parameters control maze-completion time by investigating the distribution of corridors, solving path length, and branching paths in the maze. Note that it is complexity, not difficulty that is investigated, this is because difficulty is relative and hard to pin down in a variable, while complexity in terms of completion-time is very specific.

Three PCG maze generation algorithms will be explored; Recursive backtracker (RecBack), Prim’s algorithm (Prims), and recursive division (RecDiv). They were chosen since they had been used in previous research by Foltin (Foltin, 2008), Kozlova et. al (Kozlova, Brown, & Reading, 2015), and Kim et. al (Kim & Crawfis, 2015), and because they offer a low implementation complexity with fast generation results, allowing iterative testing to be performed.

During the literature review on the topic of PCG Maze algorithms, it was challenging to find formal sources that uses RecBack, RecDiv and/or Prims in a similar context, many scientific papers on the subject almost solely rely on online sources for reference on maze generation algorithms (ex: (Foltin, 2008), (Kozlova et al., 2015), (Bae, Kim, Lee, Kim, & Na, 2015), (Kim & Crawfis, 2015). This situation offers the opportunity to contribute to this area of research.

1.2 Research Questions

- **RQ1:** What is the difference in complexity between the maze generation algorithms recursive backtracker (RecBack), Prim’s algorithm (Prims), and recursive division (RecDiv), in terms of completion time, when solved using a depth-first-search (DFS) algorithm?

- **RQ2:** What connection is there between completion time/complexity, and the distribution of branching paths, corridors, and length of the path traversed by the DFS?

Path length is explored because previous contributions by Kozlova et. al (Kozlova et al., 2015) and Foltin (Foltin, 2008) uses it as a parameter for evaluation.

Completion time is explored both because the author hypothesizes that it could be a good metric for complexity, and since Foltin (Foltin, 2008) uses a similar parameter (albeit he uses human test subjects to solve mazes).

In addition, corridor distribution and branches are explored because the author observes it can be a potential feature that could affect completion time.

1.3 Thesis Layout

This thesis consists of 6 chapters; Introduction, Background, Method, Results, Analysis and Discussion, Conclusions and Future Work, Appendix A and B. Background contains the related work and theoretical background sections, which offers a comprehensive overview of maze algorithms, graph theory, and similar research. Method
1.3. Thesis Layout

presents the tools used for the implementation, explains what is measured, and
presents the experiment procedure. Results presents and explains the gathered data,
and exposes patterns discovered in the data. Analysis and Discussion discusses the
gathered data and patterns found in it. Conclusions and Future work uses some of
the found patterns to answer RQ1 and RQ2, and gives suggestions to keep exploring
the field of PCG Maze research. Appendix A presents pseudo-code for the maze
generation algorithms. Appendix B presents code used in the implementation used
in this study.
Chapter 2

Background

This chapter will provide the theoretical background needed for the maze algorithms, as well as present related work in this field.

2.1 Theoretical Background

The PCG maze algorithms used in this thesis relies on graph theory as their foundation. This section will present the necessary concepts.

2.1.1 Graph

The graph is one of the pillars of graph theory. A graph consists of a vertex set, an edge set, and a mapping function called relation of incidence. This function states that each edge has two vertices. Another graph-attribute is connectivity which is used to distinguish between connected and disconnected graphs. In a connected graph all vertices are connected, which means that there is a path (unbroken chain of edges) between all pairs of vertices (Foltin, 2008). Graphs are visualized in Figure 2.1.

Graph Illustration

Figure 2.1: Red dots are vertices, black lines are edges. (a) displays a disconnected graph, where vertex 1 is not connected to the other vertices (i.e. a broken chain). (b) displays a connected graph, where all vertices are connected by an unbroken chain of edges.
2.1.2 Spanning Tree

A tree in graph theory is a connected graph with no cycles (Gross, Yellen, & Zhang, 2014), a cycle being defined by Black as "A path that starts and ends at the same vertex and includes at least one edge." (E. Black, 2017)\(^1\). A tree is considered a spanning tree when it includes all the graph vertices ("spans" across all vertices) (Foltin, 2008, p.13).

2.1.3 Weighted Graph

A weighted graph has a weight associated with each edge (E. Black, 2003). A common use of weight is having it represent the length of the edge. For instance classical Prim’s algorithm searches for the minimum(-weight) spanning tree in a weighed graph where weight represents edge length (E. Black, 2014).

2.1.4 Greedy Algorithm

A greedy algorithm takes the best local solution while finding an answer. The final answer might not always be the most optimal solution, but the strength of a greedy algorithm is that it finds a solution quickly (E. Black, 2005).

2.1.5 Procedural Content Generation

PCG is used in several games, and has many uses. Some examples of contemporary use is to algorithmically generate game content like the loot drops in Action Role Playing Games (ARPGs)\(^2\), or nearly infinite voxel levels like in Minecraft \(^3\).

There are three different ways to procedurally generate content according to Togelius et. al: constructive, search based, and simple generate-and-test, where constructive PCG techniques generates all content in one pass. This thesis will focus on constructive methods, since they are not as complex to implement, and generally fast, enabling generation during runtime, providing faster iteration speeds compared to other PCG approaches (Togelius, Yannakakis, Stanley, & Browne, 2011).

A type of content particularly well suited to procedural generation is mazes. This is because procedural maze algorithms can generate large varieties of different types of mazes without sacrificing quality (Kozlova et al., 2015). Three widely known maze generation algorithms are randomized Prim’s algorithm (Prims) (Foltin, 2008) (Buck & Carter, 2015) based on Prim’s Algorithm (Prim, 1957), Recursive Division algorithm (RecDiv) (Kozlova et al., 2015) (Buck & Carter, 2015), and Recursive Backtracker (RecBack) (Foltin, 2008) (Buck & Carter, 2015) based on the depth-first-search technique (Gross et al., 2014) (Kozlova et al., 2015). Each algorithm produces perfect mazes, and they either produce mazes by wall adding or passage carving. RecDiv is an example of a wall adder, while randomized Prim’s (referred to simply as Prims for convenience) algorithm can be made to do either (Foltin, 2008).

---

\(^1\)If drawn on paper, it’s edges would form a cyclical, or "round" pattern.


2.1. Theoretical Background

Both RecDiv and Prims are graph based maze algorithms. Essentially graph based algorithms create mazes by constructing a spanning tree.

2.1.6 Perfect Maze

A perfect maze is, according to Kim et. al, comprised of a spanning tree graph, a start location, and an end location (Kim & Crawfis, 2015).

2.1.7 Randomized Prim’s Algorithm

Prim’s algorithm is a greedy algorithm that finds a minimum-length spanning tree in a weighted graph (Gross et al., 2014). The randomized version of Prim’s algorithm does not need a weighted graph, since instead of it choosing the node with least weight, it is made to instead choose a random node. See Appendix A.1 for pseudocode of Prim’s algorithm.

Prim’s algorithm in its original form is not usable as a means to generate random mazes since it lacks a random factor. In Prim’s algorithm the weight is what determines the frontier edge (see step 4b in Appendix A.1), so therefore it will be replaced by a random factor as seen in the pseudocode for Randomized Prim’s Algorithm in Appendix A.2.

See Figure 2.2 for illustration of Randomized Prim’s Algorithm maze creation.

![Randomized Prims Algorithm](image)

Figure 2.2: Randomized Prim’s algorithm example. This algorithm requires only two data set structures of storing cells: a set of in-cells (grey color), and a set of frontier cells (yellow color).
Chapter 2. Background

2.1.8 Recursive Backtracker / Depth-First-Search Algorithm

The Recursive Backtracker algorithm is based on the depth-first-search (DFS) algorithm widely used in computer science (Foltin, 2008)(Gross et al., 2014)(Joshi, 2017). A DFS visits all vertices of a graph. Initially all vertices are marked new, when a vertex is visited it is marked old. Depth first search explores the graph by visiting a new-marked vertex, and then recursively call itself on all of the vertex’s adjacent new-marked vertices(Gross et al., 2014). When a DFS cannot find any adjacent new-marked vertex, the algorithm "backtracks" up the recursive call chain until it finds a vertex that has adjacent new-marked vertices, this process repeats itself until all vertices in the graph has been visited. See Appendix A.3 for pseudocode of the Recursive Backtracker algorithm. See Figure 2.3 for illustration of Recursive Backtracker maze generation.

![Recursive Backtracker Algorithm](image)

Figure 2.3: Recursive Backtracker algorithm example. Yellow points depict cells where the new walk started from.

2.1.9 Recursive Division Algorithm

Recursive backtracker and Prim’s algorithm are graph-based.

Recursive Division algorithm treats the maze as a fractal - a repeating pattern whose component parts are similar to the whole. It splits the grid into two sub-grids, putting a wall between them and a single opening in it, linking the sub-grids together, and then recursively repeats this process until the desired resolution has been produced. In the end a maze has been created(Buck & Carter, 2015). The fractal nature of the algorithm gives it the ability to potentially make infinitely complex mazes. See Appendix A.4 for pseudo-code of the Recursive Division algorithm.
Kozlova et. al investigates the path length of three two-dimensional mazes generated by Prim’s algorithm, the Depth First Search algorithm (DFS), and the Recursive Division algorithm. The purpose behind this is to better understand the expressive control and ability of the different maze algorithms. Generating mazes on a 16x11 grid, they found that that DFS generally generated the longest paths, followed by Recursive Division, and last Prims. Using a pairwise two tailed t-test they also found that the differences between the means was statistically significant. In their conclusion they advise using the DFS algorithm as a base for new maze representations, since it generates long hallways(Kozlova et al., 2015). Two things were found unclear; they never explicitly say what a path is, or how it is measured, or why they go by the assumption that long hallways is a good feature of a maze. A guess is that they define a path as a possible solution path, and the length as number of traversed nodes. They mention long hallways in passing "[...] produces mazes that have long and winding passes, which seems to be a good feature for a maze." they they cite Ashlock et. al. that wrote a paper investigating search-based procedural generation of mazes(Ashlock, Lee, & McGuinness, 2011). Ashlock et. al never explicitly state that long hallways is a good feature of mazes, they merely say that some fitness functions favor long, winding paths. Kozlova et. al probably took this statement and assumed that a fitness-function favoring a certain type of result means that that result is inherently better. This is not necessarily the case, since they could just as
well have used a fitness function favoring short paths.

Foltin made an exploratory study of human interaction with mazes. He made an implementation in the form of a Java applet that participants used to solve mazes pre-generated by the Randomized Prim’s Algorithm, Randomized Kruskal's algorithm, Hunt and Kill algorithm, and Recursive Backtracker. He measured successful attempts (SA), successful attempts count (SAC), average solution time (AST), total time (TT), solution path (SP), shortest solution path (SSP), average number of moves (ANM), and minimum number of moves (MNM). Users of the applet were awarded "competition points" when finishing mazes to incentivize users to use the app. Foltin states that this point system likely drew players toward completing quick-to-solve mazes to gain the maximum amount of points. The results of the randomized mazes statistics implied that users heavily tilted towards finishing mazes with the lowest amount of ANM and AST of point-awarding mazes. The most popular maze to solve, with the lowest AST and ANM but by far highest SAC was Randomized Prim’s, and the least popular maze to solve was Recursive Backtracker (even though it did not have the highest AST or ANM). He measured something he called "got lost ratio", which expresses how many unnecessary moves a human took in the maze.

Kim et. al (Kim & Crawfis, 2015) proposed a search-based framework for generating perfect mazes, with the purpose of giving the designer a tool for finding the "best" perfect maze, where "best" is up to the subjective opinion of the designer to decide. What drives a search-based system is a fitness function. A fitness function takes a set of user-determined variables as input, and then outputs a fitness number. The fitness function is run through a search-space (a list of sample content) where it samples fitness variables, and then the sample that outputs the highest fitness-number "wins", and it gets selected as output content.

They used several variables as inputs to the evaluation function. Of note is that they had defined several types of dead-ends, and useful concepts such as decision cell that refers to cells with more than two passages connected to it, which might also be conceptualized as "branches". The variables used in the proposed framework were:

**Metrics Per Spanning Tree:**

- Number of cells
- Number of edges
- Percentage of junctions
- Percentage of crossroads

It is somewhat unclear how an edge is recorded, however one could make the assumption that it is the number of nodes in each edge of the grid. A junction is decision-cell with three paths connected to it, which means it has just one wall, while a crossroad is a decision-cell with four paths connected to it, meaning it has no walls. The percentage of junctions and crossroads is relative to the total number of nodes in the grid.

**Metrics per solution path:**
2.2. Related Work

- Percentage of dead-ends
- Percentage of alcoves
- Percentage of forward dead-ends
- Percentage of backward dead-ends
- Percentage of turns on the solution path
- Percentage of junctions on the solution path
- Percentage of cross roads on the solution path
- Percentage of decisions on the solution path
- Relative length of the solution path

An alcove is a dead-end with no turns or decisions. It has a straight short path. A forward dead-end looks like it is progressing towards the end-point, or continues straight from the solution path. A backward dead-end turns away from the solution path and away from the end point.

The search-space was created by generating several mazes with either the randomized recursive backtracker or the randomized Prim’s algorithm.
Chapter 3
Method

An implementation will be used as the method to experimentally evaluate RQ1 and RQ2.

The selected maze algorithms must be able to fulfill these constraints before they can be used:

- Perfect maze.
- Two dimensional.
- Follows rectangular grid; the depth-first-search algorithm will follow a grid, hence the maze needs to follow the same grid to become navigable by the depth-first-search.
- Controllable entrance / exit; possibility to intentionally set the entrance and exit coordinates of the generated maze.
- Solvable; the exit must be reachable from the entrance
- Determinism; possibility to intentionally generate the exact same maze. So that experiment conditions can be replicated with the possibility to perhaps try different maze generation algorithms than those used in this experiment. Using a pseudo-random generator taking a seed-number as input is a method of achieving replicable "randomness".

The algorithms will generate mazes of different resolutions; 16x16, 32x32, 64x64, 128x128, 256x256. Those resolutions are chosen to see how an exponential increase in maze size affects the overall results of the test. Initial testing showed that resolutions exponentially higher than 256x256 were not possible due to memory limitations. This implementation allocated the resources for the algorithms in the RAM memory, and when trying resolutions higher than 256x256 (specifically tested 512x512), the memory offered by the testing machine (16 GB) was not enough. The resolutions are symmetrical in both axis to remove its ability to affect testing results.

3.1 Development Tools

The test itself does not need a graphical representation, however it is useful for debugging purposes as a means to visually verify that the maze algorithms produce the intended results. The maze is visualized in 3D, by spawning and placing simple
3D models where the maze has walls, on the traversed path, and on the solving path. It is visualized in 3D because of author preference and familiarity. The game engine Unity (version 2017.4.1f1) \(^1\) is used as the development environment since it offers a convenient way to spawn 3D-models from code, and good integration with C#. No visualizations of the mazes are generated for the main test in order to avoid making the visualization a potential factor that could affect the final results.

The code written for this thesis is written in C# \(^2\) since it is an interpreted scripting language with no compile times, which helps improve implementation speed. The IDE used for writing the code was Microsoft Visual Studio Professional 2017. The 3D-models used for the 3D visualization of the maze were produced using Autodesk Maya 2017.

### 3.2 Implementation Details

The algorithms were implemented to execute using a single CPU thread, because the author prefers this type of architecture. The speed of the test is not prioritized, the accuracy of the test is.

Several of the maze algorithms (RecBack & Prims) had to be modified so that they do not overload the computer’s stack, causing the “stack overflow” error\(^3\). The depth-first-search algorithm used for solving the mazes also needed this modification. This slowed down the maze solver, but since it is a common parameter across all maze algorithms, this should not affect the overall results. The modification itself is accomplished by simulating the recursivity with the use of the C# "Stack" container class, a data-type for storing the state of the node, and a while-loop. Example code can be found in Appendix B.1.

The complete source code for the unity project can be accessed at the author’s GitHub \(^4\).

The source code for the recursive backtracker algorithm can be found in Appendix B.2, the recursive division algorithm in Appendix B.3, and Prim’s algorithm in Appendix B.4.

### 3.3 DFS Solver and Completion Time

The mazes will be solved using a depth first search, since the author thinks it might simulate how a human would traverse the maze better than A*, Dijkstra’s, or breadth-first search. The reasoning behind this is based on the hypothesis that a person would walk down a path until either the exit, or a dead end is reached. If a dead end is reached, a person would walk back and try an untried path, and repeat this process until the end is reached. DFS simulates this behavior, but it does not

---

\(^1\) Unity Technologies, Unity3D. [https://unity3d.com/](https://unity3d.com/) (accessed 27-04-2018)


3.4 Branches

A branch is defined by the number of neighboring path-nodes, where the maximum amount of branches is determined by the number of allowed path neighbors. In a grid where only orthogonal paths are allowed, a branch can at max have 4 neighbours, whereas if vertical paths are allowed a branch can have at max 8 neighbors. However the minimum amount of neighbours for a branch to be considered a branch is always 1. The grid used in this thesis only allows orthogonal paths, so a branch can only have 4 neighbours. Branches will be recorded per maze-node of a generated maze, which means that the total number of branches will be the same as the total number of nodes in the maze which is the same as the maze resolution (ex. 16x16 = 256). Examples of branches are found in Figure 3.1. Branches as defined here are similar to the "decision cells" used by Kim et. al (Kim & Crawfis, 2015). They define a decision cell as "A cell in which there are more than two passages connected to it.", which would make the 3 and 4 branches decision cells. However they do not explicitly measure 2-branches/paths, and they do not measure dead ends in the same way as 1-branches, since they also measure the path leading up to the dead end.

![Definition of Branches](image)

Figure 3.1: Shows how branches are defined. A shows 1 branch, which is a dead end. B shows 2 branches, which is a normal path / corridor / road. C shows 3 branches, which is a fork. D shows 4 branches, which is a crossroad.
3.5 Path Length

Solving path length is the number of nodes in the found path to the exit. Traversed nodes is the total number of nodes that were visited (including the solving path length). In the code path length data is recorded by the solver.

3.6 Corridor Distribution

Corridor length will be measured as "the number of nodes in a straight chain of nodes", see Figure 3.2 for an illustration of this definition.

Definition of Corridor Length

![Corridor Length Diagram]

Figure 3.2: Shows how corridor length will be calculated. Green rectangles encloses corridors of length 2. Blue rectangle encloses corridors of length 3. Red rectangle encloses corridors of length 6.

3.7 Experiment Procedure

1. A computer with the following specifications was used:
   - Processor: Intel Xeon E5 1620 V3 3.5 GHz 10MB
   - Graphics Card: NVIDIA GeForce GTX 1080
   - Hard Drive: SK Hynix SC308 SATA 512GB Solid State Drive
   - RAM: 16 GB DDR4
   - OS: 64-bit Windows 10 Education

2. For each maze algorithm:
   
   (a) Make sure no interfering background processes are run on the testing computer. Do this by shutting down any major applications such as web browsers using the windows task manager.
   
   (b) Generate 100 mazes for each grid resolution.
   
   (c) Solve them one at a time using the depth-first-search algorithm, automatically gather the completion time, path-length, corridor distribution, and branch distribution, and write them to a Comma-separated values-file (file-extension .csv).

3. Pass the CSV-file into Excel for statistical analysis to answer RQ1 and RQ2.
All data tables presented are part of the same data set, which have been split up into the three component parts; path length, corridor distribution, and branch distribution. Each will be presented separately, and is an average of the raw data from 100 tests per resolution per maze algorithm.

4.1 Maze Appearance

Figures 4.1a, 4.1b , and 4.1c shows representative solutions obtained by the different maze algorithms. Several things stand out when observing the maze solutions:

- The maze walls of RecDivs look orderly, like squares stacked next to each other. This is an effect of the recursive nature of the algorithm, since it does actually divide the maze grid into several "boxes".

- RecBacks solution path covers much of the maze, it is also the maze algorithm with the longest solving path. Its solution path is more winding compared to the other algorithms.

- Prims solution path almost goes straight to the goal, it is also the maze algorithms with the shortest solving path.
Chapter 4. Results

Generated Mazes Representative of Each Algorithm

Figure 4.1: 32x32 mazes generated by the algorithms. (a) displays a maze generated by RecBack, (b) shows a maze generated by RecDiv, (c) illustrates a maze generated by Prims. Green nodes are part of the solution path, while red nodes are paths that lead to dead ends. Bottom left is the start/entrance of the maze solver, top right is the goal/exit.

4.2 Completion Time

Table 4.1 and Figure 4.3 shows an overview of the completion times. Figures 4.2a, 4.2b, 4.2c, 4.2d, and 4.2e graphically illustrates the completion times, which exposes these patterns:

- At resolution 16x16: RecDiv has the longest completion time, RecBack has intermediate completion time, and Prims has the shortest completion time.

- At all other resolutions: RecBack has the longest completion time, RecDiv has intermediate completion time, and Prims has the shortest completion time.
4.2. *Completion Time*

**Comparison of Completion Time By Resolution**

(Images of bar graphs showing completion time for different maze algorithms and resolutions)

Figure 4.2: *Y axis is expressed in milliseconds, signifying the time it took for the solver to find the goal. X axis shows grid resolutions for each maze algorithm.*
Figure 4.3: Y axis is expressed in milliseconds, signifying the time it took for the solver to find the goal. X axis shows grid resolutions for each maze algorithm.

Table 4.1: Completion time (in milliseconds) per resolution.
4.3 Path Length

Table 4.2: An overview of the average path length data, sorted by resolution. Completion time is in milliseconds. Solving Path (%) is Solving Path Nodes divided by Traversed Nodes, which is the proportion of Traversed Nodes that is solving path nodes. Solving path nodes is the number of nodes that is part of the completion path. Traversed nodes is the total number of nodes "walked" or "traversed" by the DFS solver, including nodes that were not part of the completion path.

Table 4.2 contains both Solving Path Nodes, and Traversed Nodes. In Figure 4.1b the sum of the green nodes is the Solving Path Nodes, while the Traversed nodes is the sum of both the red and green nodes combined. Table 4.2 shows that at higher resolutions, the RecBack has significantly higher completion time than the other algorithms. RecBack also has the highest amount of traversed nodes and solving path nodes, meanwhile Prims has the lowest completion times, and the lowest amount of solving path nodes and traversed nodes. RecDiv comes in between Prims and RecBack in all fields, being statistically closer to Prims than RecBack.

Figure 4.4 shows that Solving Path (%) decrease in a linear manner when resolutions are increased.
Path Length Increase

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>16x16 - 32x32 (%)</th>
<th>32x32 - 64x64 (%)</th>
<th>64x64 - 128x128 (%)</th>
<th>128x128 - 256x256 (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecDiv</td>
<td>524.52</td>
<td>394.77</td>
<td>373.81</td>
<td>392.42</td>
</tr>
<tr>
<td>RecBack</td>
<td>477.95</td>
<td>399.73</td>
<td>426.07</td>
<td>363.08</td>
</tr>
<tr>
<td>Prims</td>
<td>462.15</td>
<td>405.04</td>
<td>397.40</td>
<td>380.38</td>
</tr>
</tbody>
</table>

Table 4.3: Increase of solving path length per resolution change. Increase is how much longer (in percent) the higher resolution solving path is compared to the lower resolution path. Increase is calculated as solving path nodes of the higher resolution divided by the solving path nodes of the lower resolution times 100.

Figure 4.4: Y axis is expressed in percentile units, that signify the proportion of Traversed Nodes that is Solving Path Nodes (values taken from Table 4.3). X axis shows grid resolutions for each maze algorithm.
4.4 Corridor Distribution

Table 4.4: An overview of the corridor distribution data, sorted by resolution. Completion time expressed in milliseconds. The Corridor (%) columns each signify the proportion of corridors in the entire maze that is of that length.

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Resolution</th>
<th>Completion Time (ms)</th>
<th>2 Length Corridor (%)</th>
<th>3 Length Corridor (%)</th>
<th>4 Length Corridor (%)</th>
<th>5 Length Corridor (%)</th>
<th>6 Length Corridor (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecDiv</td>
<td>16x16</td>
<td>0.59</td>
<td>5B.95</td>
<td>ak.65</td>
<td>10.k5</td>
<td>k.P9</td>
<td>1.51</td>
</tr>
<tr>
<td>RecDiv</td>
<td>16x16</td>
<td>0.51</td>
<td>5a.1P</td>
<td>a6.64</td>
<td>10.41</td>
<td>5.BP</td>
<td>a.91</td>
</tr>
<tr>
<td>732 8</td>
<td>16x16</td>
<td>0.3B</td>
<td>51.40</td>
<td>ak.a0</td>
<td>11.40</td>
<td>5.B5</td>
<td>a.B8</td>
</tr>
<tr>
<td>RecDiv</td>
<td>PaxPa</td>
<td>a.P4</td>
<td>5B5P</td>
<td>ak.a9</td>
<td>10.8P</td>
<td>k.P5</td>
<td>1.6a</td>
</tr>
<tr>
<td>Recr mcs</td>
<td>PaxPa</td>
<td>P.k6</td>
<td>5a.5a</td>
<td>a5.9P</td>
<td>11.5k</td>
<td>5.1a</td>
<td>a.54</td>
</tr>
<tr>
<td>732 8</td>
<td>PaxPa</td>
<td>1.5a</td>
<td>5a.5a</td>
<td>ak.19</td>
<td>11.5P</td>
<td>5.BP</td>
<td>a.4a</td>
</tr>
<tr>
<td>RecDiv</td>
<td>6kx6k</td>
<td>1a.a6</td>
<td>5B61</td>
<td>ak.kP</td>
<td>10.56</td>
<td>k.a5</td>
<td>1.6a</td>
</tr>
<tr>
<td>Recr mcs</td>
<td>6kx6k</td>
<td>a.a1k</td>
<td>5a.4B</td>
<td>ak.09</td>
<td>11.5e</td>
<td>5.6k</td>
<td>a.4a</td>
</tr>
<tr>
<td>732 8</td>
<td>6kx6k</td>
<td>B.14</td>
<td>5a.4B</td>
<td>ak.09</td>
<td>11.56</td>
<td>5.6k</td>
<td>a.4a</td>
</tr>
<tr>
<td>RecDiv</td>
<td>14x14</td>
<td>56.a6</td>
<td>5B61</td>
<td>ak.kP</td>
<td>10.6k</td>
<td>k.19</td>
<td>1.81</td>
</tr>
<tr>
<td>Recr mcs</td>
<td>14x14</td>
<td>aB1.14</td>
<td>5P1P</td>
<td>a6.a6</td>
<td>11.kk</td>
<td>5.0k</td>
<td>a.6a</td>
</tr>
<tr>
<td>732 8</td>
<td>14x14</td>
<td>PPkPa</td>
<td>5a.6k</td>
<td>ak.a4</td>
<td>11.kk</td>
<td>5.59</td>
<td>a.6k</td>
</tr>
<tr>
<td>RecDiv</td>
<td>a56x56</td>
<td>P6a.1k</td>
<td>5B6P</td>
<td>ak.k0</td>
<td>10.65</td>
<td>k.14</td>
<td>1.8a</td>
</tr>
<tr>
<td>Recr mcs</td>
<td>a56x56</td>
<td>6kko.0k</td>
<td>5PaP</td>
<td>a6.a1</td>
<td>11.kk</td>
<td>5.05</td>
<td>a.61</td>
</tr>
<tr>
<td>732 8</td>
<td>a56x56</td>
<td>16k.45</td>
<td>5P0a</td>
<td>ak.aP</td>
<td>11.kk</td>
<td>5.56</td>
<td>a.6k</td>
</tr>
</tbody>
</table>

There are corridor lengths higher than length 6 presented in this section, but the appearance of such corridors was a rare occurrence, therefore those were left out of the presented data. Table 4.4 shows that the corridor distribution of each maze algorithm does not differ much per resolution change. However there is a slight difference in corridor distribution between the algorithms.

Observing Figures 4.5a, 4.5b, 4.5c, 4.5d, and 4.5e resulted in Table 4.5, which compares the corridor lengths of each algorithm to each other. Those comparisons exposes these patterns:

- Prims, the algorithm with the fastest completion time has the fewest 2 and 3 length paths, but the highest proportion of 4, 5 and 6 length paths.

- RecDiv with the intermediate completion time has the highest proportion of 2 length paths, intermediate proportion of 3 length paths, and the lowest proportion of 4, 5 and 6 length paths.

- RecBack, with the slowest completion time has the lowest proportion of 2 length paths, highest of 3 length paths, but a intermediate proportion of 4, 5 and 6 length paths.
## Path Length Comparison

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Length (nodes)</th>
<th>H</th>
<th>M</th>
<th>L</th>
<th>Maze Type</th>
<th>Length (nodes)</th>
<th>H</th>
<th>M</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecDiv</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
<td>RecDiv</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecDiv</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
<td>RecBack</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecDiv</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
<td>Prims</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecDiv</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
<td>RecDiv</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecDiv</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
<td>RecBack</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecBack</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
<td>Prims</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecBack</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
<td>RecDiv</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecBack</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
<td>RecBack</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecBack</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
<td>Prims</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecBack</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
<td>RecDiv</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prims</td>
<td>2 X</td>
<td></td>
<td></td>
<td></td>
<td>RecBack</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prims</td>
<td>3 X</td>
<td></td>
<td></td>
<td></td>
<td>Prims</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prims</td>
<td>4 X</td>
<td></td>
<td></td>
<td></td>
<td>RecDiv</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prims</td>
<td>5 X</td>
<td></td>
<td></td>
<td></td>
<td>RecBack</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prims</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
<td>Prims</td>
<td>6 X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Blue is sorted by algorithm, green is sorted by corridor length. Shows a relative comparison between the corridor lengths of the different maze algorithms. H = highest relative proportion, M = intermediate relative proportion, L = lowest relative proportion.
4.5 Branch Distribution

Table 4.6 displays an overview of the branch distribution data. It is observed that branch distribution does not change much for the algorithms between resolutions. However, there is a difference in the proportions of branches between the algorithms. Observing figures 4.6a, 4.6b, 4.6c, 4.6d, while comparing the maze algorithms to each other resulted in Table 4.7, which helped expose some patterns:

Figure 4.5: Length is stated in the title of each sub-figure. Shows distribution of corridors of stated length across resolutions. Y axis expressed in percentile units, which signifies the proportion of corridors that is of stated length. Proportion is calculated from dividing the number of corridors of stated length by the total number of corridors in the maze. X axis shows grid resolutions for each maze algorithm.
• RecBack, with the highest completion time, has the highest distribution of 2-branches, but the lowest distribution of every other branch type, with 4-branches close to 0% distribution.

• Prims is the opposite to RecBack, with the lowest completion time, lowest distribution of 2-branches, and the highest distribution of all of the other branch types.

• RecDiv, with medium completion time has medium distribution across all branch types.

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Resolution</th>
<th>Completion Time (ms)</th>
<th>1 Branches (%)</th>
<th>2 Branches (%)</th>
<th>3 Branches (%)</th>
<th>4 Branches (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecDiv</td>
<td>16x16</td>
<td>0.59</td>
<td>Ba.69</td>
<td>kP.0r</td>
<td>B1.6k</td>
<td>B.6r</td>
</tr>
<tr>
<td>RecDiv</td>
<td>16x16</td>
<td>0.51</td>
<td>10.a0</td>
<td>a9.k9</td>
<td>9.a1</td>
<td>0.11</td>
</tr>
<tr>
<td>73 2 8</td>
<td>16x16</td>
<td>0.ka</td>
<td>r.k.a1</td>
<td>r.6.P0</td>
<td>Br.0k</td>
<td>5.k5</td>
</tr>
<tr>
<td>RecDiv</td>
<td>8 Bkr B</td>
<td>8.r P</td>
<td>Ba.r P</td>
<td>kP.1P</td>
<td>B1.a1</td>
<td>B.a9</td>
</tr>
<tr>
<td>RecDiv</td>
<td>8 Bkr B</td>
<td>r.k6</td>
<td>10.r B</td>
<td>a9.a0</td>
<td>9.Pr</td>
<td>0.15</td>
</tr>
<tr>
<td>73 2 8</td>
<td>8 Bkr B</td>
<td>1.5B</td>
<td>r.5.11</td>
<td>r.5.P0</td>
<td>Br.B9</td>
<td>5.P1</td>
</tr>
<tr>
<td>RecDiv</td>
<td>6kx6k</td>
<td>1.B.B6</td>
<td>Ba.k0</td>
<td>kP.0r</td>
<td>B1.aP</td>
<td>B.aP</td>
</tr>
<tr>
<td>RecDiv7</td>
<td>6kx6k</td>
<td>B.B1.k</td>
<td>10.06</td>
<td>P0.0P</td>
<td>9.a0</td>
<td>0.16</td>
</tr>
<tr>
<td>73 2 8</td>
<td>6kx6k</td>
<td>a.1P</td>
<td>r.5.k6</td>
<td>r.5.19</td>
<td>Br.r0</td>
<td>6.06</td>
</tr>
<tr>
<td>RecDiv</td>
<td>18Px1BP</td>
<td>56.6B</td>
<td>Ba.r k</td>
<td>kP.06</td>
<td>B1.P5</td>
<td>B.a9</td>
</tr>
<tr>
<td>RecDiv7</td>
<td>18Px1BP</td>
<td>Ba1.1P</td>
<td>10.0B</td>
<td>P0.15</td>
<td>9.65</td>
<td>0.1P</td>
</tr>
<tr>
<td>73 2 8</td>
<td>18Px1BP</td>
<td>r.r.r B</td>
<td>r.5.59</td>
<td>r.k.95</td>
<td>Br.rk</td>
<td>6.1B</td>
</tr>
<tr>
<td>RecDiv</td>
<td>B56xB56</td>
<td>6B.1k</td>
<td>Ba.r 9</td>
<td>ka.9P</td>
<td>B1.P6</td>
<td>B.a6</td>
</tr>
<tr>
<td>RecDiv7</td>
<td>B56xB56</td>
<td>6,kk0.0k</td>
<td>9.9a</td>
<td>P0.8k</td>
<td>9.61</td>
<td>0.1P</td>
</tr>
<tr>
<td>73 2 8</td>
<td>B56xB56</td>
<td>16k.55</td>
<td>r.5.aB</td>
<td>r.k.a6</td>
<td>Br.rr</td>
<td>6.19</td>
</tr>
</tbody>
</table>

Table 4.6: An overview of the branch distribution data, sorted by resolution. Completion time in milliseconds. Each Branches (%) column signifies the proportion of nodes with that number of branches. Note that the sum of all branch columns in one row is 100, since the sum of all branches is 100% of the branches. Further note that the total number of branches is equal to the total number of nodes in the maze (which is the same as the resolution of the maze), since each node has branches.
Table 4.7: Blue is sorted by algorithm, green is sorted by branches. Shows a relative comparison between the branches of the different maze algorithms. H = highest relative proportion, M = intermediate relative proportion, L = lowest relative proportion.

Figure 4.6: The compared branch type is stated in each sub-figure. Y axis is expressed in percentile units, signifying the proportion of all branches that are of the stated type. X axis shows grid resolutions for each maze algorithm.
Chapter 5

Analysis and Discussion

This chapter will present, explain, and argue for and against patterns found in the results of each data set in turn.

5.1 Completion Time

The data presented in Section 4.2 shows that in all resolutions, except 16x16, RecBack is slowest, Prims is fastest, and RecDiv is intermediate.

The order of complexity of the maze algorithms coincides with the order of path length in the work of Kozlova et. al (Kozlova et al., 2015), who calculated the mean of path lengths of the RecDiv, Prims, and RecBack (called depth first search) algorithms. Since path length seems to correlate with complexity (argumentation found in Section 5.2), their findings should increase the confidence in the answer to RQ1.

The order of complexity of the maze algorithms also partially coincides with the findings of Foltin (Foltin, 2008). Foltin investigated RecDiv and Prims (among some other algorithms) and recorded the average completion time of humans solving those mazes. The data shows that RecBack took longer for humans to solve than Prims. It implies that they found RecBack harder to solve, which implies increased complexity. Since Foltin did not use RecDiv and he used human test subjects instead of a solver algorithm, his results does not directly correlate as much as Kozlovas. However his results does imply that the results found in this thesis may also be applicable to human test subjects.

It is uncertain what exactly makes the completion time of RecBack so much higher than Prims and RecDiv. One could think that since the completion time is the time it takes for the DFS solver to reach the goal, each traversed node takes time for the solver to traverse. Therefore the traversed nodes should be directly correlated with completion time. However it is not explained by the traversed nodes, since RecDiv on resolution 256x256 has only traversed 15% more nodes than RecDiv, which does not explain the 1800% higher completion time. However the amount of solving-path nodes of RecBack is 5.73 times higher than RecDiv, which could mean that it is part of the answer. Or perhaps the completion time difference can be found as a combination of solving path nodes and traversed nodes, or in a statistic not taken into account. It could be the case that it is because of a software error, however it is unlikely since the change in traversed nodes, solving path nodes, and completion time is too consistent with previous contributions. Previous contributions were looked through to find the answer to the big difference in completion time. Foltin (Foltin,
2008) measured the average completion time of maze resolutions similar to the 16x16, 32x32, and 64x64 resolutions used in this thesis. His completion time increase seem to approximately coincide with the ones found between resolutions 16x16, 32x32, and 64x64 found in Figure 5.2 (however his increase is smaller, and less even). However it should be noted that his resolutions are not the same as in this thesis, and the solving time he gathered was that of humans, not a DFS. No other contributions measuring the completion time of mazes of differing resolutions has been found, which makes answering the completion time increase difference the task of future research.

While observing Table 4.1, of note is that the difference in complexity seems to get smaller as maze resolutions get lower. This might occur since as the maze grid gets smaller, the number of possible maze variations inside that grid gets smaller, and therefore outputs of the algorithms seem more and more similar. So the difference in complexity between the maze algorithms seems to be directly proportional to the resolution of the maze grid.

![Prims Completion Time Increase](image)

**Figure 5.1:** Y axis is change/increase, described in percentages as how the completion time of the higher resolution compares to the completion time of the lower resolution. Change/Increase is calculated as larger resolution completion time divided by smaller resolution completion time times 100. X axis shows grid resolutions for each maze algorithm.
5.2 Branches, Path Length, and Corridors Effect On Completion Time

To draw conclusions about potential connections to complexity, patterns offering clear opposites was searched for. Such patterns makes it easier to predict what effect

Figure 5.2: Y axis is change/increase, described in percentages as how the completion time of the higher resolution compares to the completion time of the lower resolution. Change/Increase is calculated as larger resolution completion time divided by smaller resolution completion time times 100. X axis shows grid resolutions for each maze algorithm.

Figure 5.3: Y axis is change/increase, described in percentages as how the completion time of the higher resolution compares to the completion time of the lower resolution. Change/Increase is calculated as larger resolution completion time divided by smaller resolution completion time times 100. X axis shows grid resolutions for each maze algorithm.

5.2 Branches, Path Length, and Corridors Effect On Completion Time

To draw conclusions about potential connections to complexity, patterns offering clear opposites was searched for. Such patterns makes it easier to predict what effect
### Average Completion Time

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Completion Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecBack</td>
<td>1,347.47</td>
</tr>
<tr>
<td>RecDiv</td>
<td>86.73</td>
</tr>
<tr>
<td>Prims</td>
<td>41.47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Completion Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecBack</td>
<td>297.30</td>
</tr>
<tr>
<td>RecDiv</td>
<td>71.50</td>
</tr>
<tr>
<td>Prims</td>
<td>42.49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Completion Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecBack</td>
<td>26.11</td>
</tr>
<tr>
<td>RecDiv</td>
<td>15.24</td>
</tr>
<tr>
<td>Prims</td>
<td>9.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Maze Type</th>
<th>Completion Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecBack</td>
<td>3.98</td>
</tr>
<tr>
<td>RecDiv</td>
<td>2.97</td>
</tr>
<tr>
<td>Prims</td>
<td>1.99</td>
</tr>
</tbody>
</table>

Table 5.1: Completion time expressed in milliseconds. (a) Presents the average completion time between all resolutions for each maze algorithm. (b) Presents the average completion time between the resolutions 16x16, 32x32, 64x64 and 128x128, for each maze algorithm. (c) Presents the average completion time between the resolutions 16x16, 32x32 and 64x64, for each maze algorithm. (d) Presents the average completion time between the resolutions 16x16 and 32x32, for each maze algorithm.
a change should induce.

5.2.1 Branches

Branch distribution patterns found in Table 4.7 indicate that completion time is directly proportional to the amount of 2-branches. A high distribution of 2-branches leads to a lower distribution of all other branch types, which in turn hypothetically would reduce the number of possible paths to the goal, making finding the solution path harder.

Figures 4.1c and 4.1a illustrate this hypothesis. Figure 4.1c which illustrates a maze generated by Prims, has a solving path that goes almost straight between start and finish. Prims as the algorithm with the least amount of 2 branches, and the highest amount of all other branch types, makes it the most interconnected maze, which makes direct paths to any point in the maze more likely to occur.

Figure 4.1a is the opposite, with the highest proportion of 2-branches, and the lowest proportion of all other branch types, making it the least interconnected maze. Because of its low interconnectivity, it’s solution path winds through the maze, covering a large portion of the maze grid.

5.2.2 Path Length

What stands out in Table 4.2 is the completion time of RecBack on resolution 256x256. The other solving path nodes are much smaller. This difference might look so large as to indicate an error in the test, but looking at Figure 4.3 and 5.4 show that the increase of solving path length has remained relatively even, which indicates that results had a low variation in the test. By observing the path length data in Table 4.2, one can discern a pattern that seems to indicate that a lower amount of solving path nodes and traversed nodes, leads to lower completion times. This is unsurprising, since traversing nodes is what takes time for a solver searching for the exit.

Mazes with relatively short solution paths seem to get solved faster. Perhaps if one thinks of the solution path as "the minimum amount of nodes to traverse to solve the maze". Solvers traversing mazes with short solution paths would then at minimum traverse fewer nodes than solvers traversing mazes with long solution paths.

Kozlova et. al (Kozlova et al., 2015) measured path lengths of the same maze algorithms as in this thesis. They also found RecBack to have the longest average path length, Prims to have the shortest average path length, and RecDiv to have intermediate path length. They used a non-uniform grid of resolution 16x12 for their maze algorithms, of a lower resolution that in this thesis (lowest 16x16), so the results, even if similar overall, are not perfectly correlated. But still it is unlikely that it is by accident that the size orders are the same.
what makes for slower completion times. Corridors lead to faster completion times, because there is no clear opposite that says time, since the differences in corridor length in Table 4.4 is at most around 3%, and it is hard to draw a conclusion around corridor distribution’s effect on completion time, since the differences in corridor length in Table 4.4 is at most around 3%, and at least below 1%. It is hard to make the case that a higher proportion of long corridors lead to faster completion times, because there is no clear opposite that says what makes for slower completion times.

5.2.3 Corridors

It is hard to draw a conclusion around corridor distribution’s effect on completion time, since the differences in corridor length in Table 4.4 is at most around 3%, and at least below 1%. It is hard to make the case that a higher proportion of long corridors lead to faster completion times, because there is no clear opposite that says what makes for slower completion times.
In this chapter, conclusions that can be drawn from the tests are stated and presented as answers to the research questions.

This study has attempted to provide level designers, developers, and game designers with information to evaluate and understand what makes some mazes more complex than others. To do this a software implementation was developed to perform tests on three different maze generation algorithms across five different maze resolutions, which gathered their solving path length, traversed nodes, corridor distribution, branch distribution, and completion time data.

**RQ1:** What is the difference in complexity between the maze generation algorithms recursive backtracker (RecBack), randomized Prim’s algorithm (Prims), and recursive division (RecDiv), in terms of completion time, when solved using a depth-first-search (DFS) algorithm?

**Answer:** Overall RecBack is more complex than RecDiv which is more complex than Prims which is the least complex. Maze resolution is directly proportional to complexity. This means that in low resolutions, the differences in complexity between the algorithms are less pronounced than at higher resolutions. The complexity of RedDiv is more similar to Prims than RecBack.

**RQ2:** What connection is there between completion time/complexity, and the distribution of branching paths, distribution of corridors, and length of the path traversed by DFS?

**Answer:**

- **Branching Paths:** Complexity seems to be directly proportional to 2-branches. Also since the proportion of one branch-type affect the proportions of other branch-types, there is a direct inversely proportional relationship between all branch types.

- **Corridor distribution:** There is not an easily observable pattern between the corridor-distribution-results of the three algorithms that seem to affect the overall completion time.

- **Path length:** Complexity seems to be directly proportional to both solving path nodes and traversed nodes.

It has in the end been observed that using solving-time as an absolute metric for complexity might not be beneficial since it heavily favors mazes with the longest...
solution-paths. The observation is that the "most complex maze" would be a maze where every node in the grid is part of the solution path, which might not be ideal for gameplay scenarios. However it could perhaps still be useful as part of a fitness-function to be used to craft mazes similarly to what Kim et. al did (Kim & Crawfis, 2015).

6.1 Future Work

For future work, other researchers could try replicating the test results which would be beneficial for the relatively sparse field of study of mazes. Perhaps a metric for complexity better than DFS solver completion time could be found. A search-algorithm specifically crafted to more closely simulate human behaviour (like having an imperfect memory, making it retread already walked paths). No search-algorithm that takes into account human has been found, therefore it would perhaps be beneficial if such an algorithm was developed, perhaps by closely observing humans solving mazes, and then translate the observed behavior to algorithms. It would provide more accurate results when measuring complexity of mazes, and could perhaps prove to have other uses. Researchers could perform a statistical analysis on the data presented to determine if the data is statistically significant. The development of a more standardized language around mazes should be looked into to nurture further research into this area. Perform tests on maze algorithms besides those used in this study to expand the body of knowledge. Further research testing the parameters investigated here could develop a more complex application which allows for direct control over the proportions of the parameters, which would allow researchers to more precisely investigate them, and to evaluate the correctness of the answers to Research Question 2. A suggestion for PCG types suited for such an application would be either an evolutionary algorithm, a grammar based one, or a search-based one. An example of a search-based application that generates mazes was developed by Kim et. al (Kim & Crawfis, 2015), who investigated ways to control maze-generation. Their paper is a good start-off point for future work in that area.

When exploring the possibility to numerically quantify the difference in complexity between the algorithms, finding numerical differences that were true for all resolutions proved unfeasible to do by simple observation. Table 5.1a shows the average completion time between all resolutions for each algorithm. Using that data would make RecDiv 2.1 times more complex than Prims, and RecBack 15.5 times more complex than RecDiv. However if the completion time for 256x256 is left out of the average completion time data the resulting data is shown in Table 5.1b. Using that data would make RecDiv 1.7 times more complex than Prims, and RecBack 4.16 times more complex than RecDiv. Tables 5.1c,5.1d shows the completion times as high resolutions are removed from the average completion time data. No clearly discernible pattern in the increasing complexity has been found in Figures 5.1, 5.2 and 5.3, except that the completion time tends to get higher as resolutions increase. In order to help explain completion time, patterns for complexity that are true for all resolutions could be found by developing an extrapolation function.
References


Appendix A

Maze Generation Algorithms

A.1 Prim’s Algorithm

Foltin (Foltin, 2008) credits Gross et. al (Gross & Yellen, 1999) for the following Prim’s algorithm pseudocode:

- **Input:** a weighted connected graph $G$.
- **Output:** a minimum spanning tree $T$.

1. Choose an arbitrary vertex $s$ of graph $G$.
2. Initialize the Prim tree $T$ as vertex $s$.
3. Initialize the set of frontier edges for tree $T$ as empty.
4. While Prim tree $T$ does not yet span $G$
   (a) Update the set of frontier edges for $T$
   (b) Let $e$ be a frontier edge for $T$ with the smallest edge-weight.
   (c) Let $v$ be the non-tree endpoint of edge $e$.
   (d) Add edge $e$ (and vertex $v$) to tree $T$.
5. Return Prim tree $T$.

A.2 Randomized Prim’s Algorithm

Foltin (Foltin, 2008) provides pseudocode for generating a Random Prim’s Algorithm Maze:

1. Choose a random cell within the maze grid (given by its width and height) and design it as a start cell.
2. Add the start cell to (by now empty) inCells set.
3. Mark cells around the start cell as frontier, i.e. add them to frontierCells set.
4. While frontierCells set is not empty:
   (a) Choose a random frontier cell $cF$ from frontierCells
Appendix A. Maze Generation Algorithms

(b) Choose a random in-cell $c_I$ adjacent to $c_F$.
(c) Add $c_F$ to inCells
(d) Mark all out-cells around $c_F$ as frontier.
(e) Add a path between $c_I$ and $c_F$ to the maze paths set.
(f) Remove $c_F$ from frontierCells set.

A.3 Recursive Backtracker Algorithm

Foltin provides pseudocode for generating a Recursive Backtracker maze (Foltin, 2008):

1. Design a randomly chosen cell as the current cell and add it to the stack.
2. IF the current cell has any unvisited neighbors THEN
   (a) randomly choose one of them and design it as the next cell
   (b) add next cell to the stack
   (c) create path between current cell and next cell
   (d) mark next cell as the current cell
3. ELSE pop a cell from the stack and design it as the current cell
4. WHILE stack is not empty
   (a) loop back to 2

A.4 Recursive Division Algorithm

Jamis Buck provides pseudocode for implementing the Recursive Division algorithm (Buck & Carter, 2015) (Buck, 2011):

1. Begin with an empty field
2. Bisect the field with a wall, either horizontally or vertically. Add a single passage through the wall.
3. Repeat step 2 with the areas on either side of the wall.
4. Continue, recursively, until the maze reaches the desired resolution.
Appendix B

Code Examples

B.1 Recursive Code

Example of recursive depth first search and stack-safe depth first search produced by Albin Karlsson (Author of thesis), written in C#. Will not work out-of-the-box since some non important functions were left out, but should provide enough guidance to an interested developer.

class Node
{
    bool isSpecial = false;
    Node[] children = new Node[4];
    //... Functions for initializing children

    void removeChild(int childIndex)
    {
        children[childIndex] = NULL;
    }
}

Node recursiveDepthFirstSearchFunc(Node currentNode)
{
    foreach (Node child in currentNode.children)
    {
        if (child.isSpecial == true)
            return child;
        else
        {
            Node result =
                recursiveSearchFunc(child);
            if (result.isSpecial == true)
            {
                return result;
            }
        }
    }
    return currentNode;
}
Node StackSafeDepthFirstSearchFunc(Stack<Node> stack)
{
    while (stack.Count > 0)
    {
        //Pop removes element from top of stack, 
        //and returns it.
        Node currentNode = stack.Pop();

        for (int i = 0; i < currentNode.children.Count; i++)
        {
            Node child = currentNode.children[i];
            if (child == NULL)
            {
                //Do nothing
            }
            else if (child.isSpecial == true)
            {
                return child;
            }
            else
            {
                //Remove child so that it’s 
                //not processed again next time
                currentNode.removeChild(i);
                //Add the current node to the 
                //stack again
                //to continue the child loop next 
                //time
                stack.Push(currentNode);
                //Child pushed to top of stack so 
                //that
                //it is processed in the next 
                //while loop iteration.
                stack.Push(child);
                //Stops the for loop
                break;
            }
        }
    }
    //No node with isSpecial == true was found.
    return Node();
}

void main()
{
    const int numNodes;
    Node nodeNetwork[numNodes];
//... Fill the node network, with only one node
//... with "isSpecial" set to true.

Node result = recursiveSearchFunc(nodeNetwork[0]);
if (result.isSpecial == true)
    print("Yippie!");
else
    print("Too bad");

Stack<Node> stack;

//Push adds an element to top of stack.
stack.Push(nodeNetwork[0]);
result = StackSafeDepthFirstSearchFunc(stack);
if (result.isSpecial == true)
    print("Yippie!");
else
    print("Too bad");

}  

B.2 Recursive Backtracker Code

Made stack-safe to prevent stack overflow. Accomplished through simulating recur-
sivity by the use of a Stack-container, and a while loop.

void recursiveBackTrack(IntVector2 pos)
{
    List<IntVector2> directionList = new List<IntVector2>();
    //initializing list of directions
    directionList.Add(NORTHDir);
    directionList.Add(SOUTHDir);
    directionList.Add(WESTDir);
    directionList.Add(EASTDir);

    shuffleDirections(directionList);

    Stack<stackState> theStack = new Stack<stackState>();
    theStack.Push(new stackState(pos, directionList));
    //set current cell as visited
    grid[pos.X, pos.Z].Visited = true;

    while (theStack.Count > 0)
    {
        stackState tempState = theStack.Pop();
IntVector2 po = tempState.pos;
grids[po.X, po.Z].Visited = true;
// Now loop through neighbours. If found
 neighbour, call self, with neighbours
 position
foreach (IntVector2 dir in
 tempState.directions)
{
    Stack<IntVector2> tempStack = new
    Stack<IntVector2> ();
    if (IsDirCoordValid(new IntVector2(po.X,
        + dir.Z].Visited == false)
    {
        if (dir == NORTHDir)
        {
            grids[po.X, po.Z].NorthWall = false;
            grids[po.X + dir.X, po.Z +
                dir.Z].SouthWall = false;
        }
        if (dir == SOUTHDir)
        {
            grids[po.X, po.Z].SouthWall = false;
            grids[po.X + dir.X, po.Z +
                dir.Z].NorthWall = false;
        }
        if (dir == WESTDir)
        {
            grids[po.X, po.Z].WestWall = false;
            grids[po.X + dir.X, po.Z +
                dir.Z].EastWall = false;
        }
        if (dir == EASTDir)
        {
            grids[po.X, po.Z].EastWall = false;
            grids[po.X + dir.X, po.Z +
                dir.Z].WestWall = false;
        }
        theStack.Push(tempState);
    } List<IntVector2> tempDirs = new
    List<IntVector2>(directionList);
    shuffleDirections(tempDirs);
    grids[po.X + dir.X, po.Z +
        dir.Z].Visited = true;
B.3 Recursive Division Code

A recursive function, it did not need to be made stack-safe. Most of it is taken from the implementation code of Buck (Buck & Carter, 2015).

```c
void recursiveDivision(int X, int Z, int width, int height, int orientation)
{
    if (width < 2 || height < 2)
        return;

    bool horizontal = (orientation == HORIZONTAL) ?
        true : false;

    int wx = X + (horizontal ? 0 : Random.Range(0, width - 2));
    int wz = Z + (horizontal ? Random.Range(0, height - 2) : 0);

    int px = wx + (horizontal ? Random.Range(0, width) : 0);
    int pz = wz + (horizontal ? 0 : Random.Range(0, height));

    int dx = horizontal ? 1 : 0;
    int dz = horizontal ? 0 : 1;

    int length = horizontal ? width : height;
    int dir = horizontal ? SOUTH : EAST;

    for (int i = 0; i < length; i++)
    {
        if (wx != px || wz != pz)
        {
            if (dir == SOUTH)
            {
```
Appendix B. Code Examples

grid[wx, wz].EastWall = true;
if (wz + 1 < gridX)
    grid[wx, wz + 1].WestWall = true;
}
if (dir == EAST)
{
    grid[wx, wz].SouthWall = true;
    if (wx + 1 < gridZ)
        grid[wx + 1, wz].NorthWall = true;
}
wx += dx;
wz += dz;
}

int nx = X;
int nz = Z;

int w = horizontal ? width : wx - X + 1;
int h = horizontal ? wz - Z + 1 : height;

recursiveDivision(nx, nz, w, h,
chooseOrientation(w, h));

nx = horizontal ? X : wx + 1;
nz = horizontal ? wz + 1 : Z;

w = horizontal ? width : X + width - wx - 1;
h = horizontal ? Z + height - wz - 1 : height;
recursiveDivision(nx, nz, w, h,
chooseOrientation(w, h));

B.4 Randomized Prim’s Algorithm Code

Made stack-safe to prevent stack overflow. Accomplished through simulating recursivity by the use of a Stack-container, and a while loop.

void primsAlgorithm(int startX, int startZ)
{
    // start from a nodes, all adjacent nodes are frontier.
    // Pick a random frontier node, add it to the maze. Now add the adjacent nodes to the new node as frontier too.
    // rinse repeat until no more frontier cells.
List<IntVector2> frontier = new List<IntVector2>();
List<mazeLoc> inCells = new List<mazeLoc>();

//Add directions
List<IntVector2> directionList = new List<IntVector2>();
//initializing list of directions
directionList.Add(NORTHDir);
directionList.Add(SOUTHDir);
directionList.Add(WESTDir);
directionList.Add(EASTDir);

//frontier.Add(new mazeLoc(startX, startZ));
inCells.Add(new mazeLoc(startX, startZ));
grid[startX, startZ].Visited = true;

foreach (IntVector2 dir in directionList)
{
    //Add valid adjacent cells to frontier.
    if ((startX + dir.X >= 0 && (startX + dir.X <
        gridX)) && (startZ + dir.Z >= 0 && startZ
        + dir.Z < gridZ))
    {
        if (grid[startX + dir.X, startZ +
            dir.Z].Visited == false && grid[startX
            + dir.X, startZ +
            dir.Z].AddedToFrontier == false)
        {
            grid[startX + dir.X, startZ +
                dir.Z].AddedToFrontier = true;
            frontier.Add(new IntVector2(startX +
                dir.X, startZ + dir.Z));
        }
    }
}

visualizePrimSolver(frontier, new IntVector2(0, 0));

while (frontier.Count > 0)
{
    //Get current frontier cell coordinate
    shuffleDirections(directionList);
    int CFI = Random.Range(0, frontier.Count);

Appendix B. Code Examples

//A) Choose random frontier cell from frontier-list
IntVector2 CFC = new IntVector2(frontier[CFI].X, frontier[CFI].Z);
grid[CFC.X, CFC.Z].Visited = true;
//frontier.Remove(frontier[CFI]);

foreach (IntVector2 dir in directionList)
{
    //Add valid adjacent cells to frontier.
    if (IsDirCoordValid(CFC, dir))
    {
        if (grid[CFC.X + dir.X, CFC.Z + dir.Z].Visited == false &&
            grid[CFC.X + dir.X, CFC.Z + dir.Z].AddedToFrontier == false)
        {
            grid[CFC.X + dir.X, CFC.Z + dir.Z].AddedToFrontier = true;
            frontier.Add(new IntVector2(CFC.X + dir.X, CFC.Z + dir.Z));
        }
    }
}

foreach (IntVector2 dir in directionList)
{
    //If cell adjacent to this frontier has already been visited, carve path to it. (only once per frontier-loop)
    if (IsDirCoordValid(CFC, dir))
    {
        if (grid[CFC.X + dir.X, CFC.Z + dir.Z].Visited == true)
        {
            carveInDirection(dir, CFC.X, CFC.Z);
            break;
        }
    }
}
frontier.RemoveAt(CFI);
B.5  Depth-first-search Algorithm Code

Made stack-safe to prevent stack overflow. Accomplished through simulating recursivity by the use of a Stack-container, and a while loop.

```csharp
solverBack fastSolveMazeReadyForVisualization(int X, int Z, List<IntVector2> SolvingPath, List<IntVector2> VisitedPath)
{
    List<IntVector2> lDirectionCheck = new List<IntVector2>();
    lDirectionCheck.Add(NORTHDir);
    lDirectionCheck.Add(SOUTHDir);
    lDirectionCheck.Add(WESTDir);
    lDirectionCheck.Add(EASTDir);

    Stack<StackStateSolver> solverStack = new Stack<StackStateSolver>();
    List<IntVector2> localSolvingPath = new List<IntVector2>(SolvingPath);
    shuffleDirections(lDirectionCheck);
    solverStack.Push(new StackStateSolver(new IntVector2(X, Z), lDirectionCheck, localSolvingPath, VisitedPath));

    while (solverStack.Count > 0)
    {
        StackStateSolver tempStackState = solverStack.Pop();
        if (tempStackState.pos.X == gridX - 1 && tempStackState.pos.Z == gridZ - 1)
        {
            tempStackState.solvingPath.Add(tempStackState.pos);
            tempStackState.visitedPath.Add(tempStackState.pos);
            solvedPath = tempStackState.solvingPath;
            triedPath = tempStackState.visitedPath;
            //Solver finished. Convert from seconds to milliseconds
            return new solverBack(true, (Time.realtimeSinceStartup - startSolveTime) * 1000f);
        }
    }
    bool isValidPath = false;
```
grid[tempStackState.pos.X, 
    tempStackState.pos.Z].SolverVisited = true;

if (tempStackState.hasBeenAdded == false) 
{
    tempStackState.visitedPath.Add(new 
    IntVector2(tempStackState.pos.X, 
        tempStackState.pos.Z));
    tempStackState.solvingPath.Add(new 
    IntVector2(tempStackState.pos.X, 
        tempStackState.pos.Z));
    tempStackState.hasBeenAdded = true;
}

foreach (IntVector2 dir in 
    tempStackState.directions) 
{
    if (IsDirCoordValid(tempStackState.pos, 
        dir)) 
    {
        if (!grid[tempStackState.pos.X + 
            dir.X, tempStackState.pos.Z + 
            dir.Z].SolverVisited)
        {
            if (dir == NORTHEst) 
            {
                if (!grid[tempStackState.pos.X + 
                    dir.X, tempStackState.pos.Z + 
                    dir.Z].SouthWall &&
                !grid[tempStackState.pos.X, 
                    tempStackState.pos.Z].NorthWall)
                {
                    isValidPath = true;
                }
            }
            if (dir == SOUTHEst)
            {
                if (!grid[tempStackState.pos.X + 
                    dir.X, 
                    tempStackState.pos.Z + 
                    dir.Z].NorthWall &&
                !grid[tempStackState.pos.X, 
                    tempStackState.pos.Z].NorthWall)
if (dir == WESTDir) {
        !grid[tempStackState.pos.X, tempStackState.pos.Z].WestWall) {
        isValidPath = true;
    }
}

if (dir == EASTDir) {
        !grid[tempStackState.pos.X, tempStackState.pos.Z].EastWall) {
        isValidPath = true;
    }
}

if (isValidPath) {
    solverStack.Push(tempStackState);
    shuffleDirections(lDirectionCheck);

    break;
}
return new solverBack(false, 0);