Procedurally Generating a Dungeon for Roguelike Games

Manan Shahi

Procedurally Generating a Dungeon for Roguelike Games

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Arts in Computer Science in the Department of Mathematical & Computational Sciences at The College of Wooster

by
Manan Shahi
The College of Wooster
2023

Advised by:

Daniel Palmer (Computer Science)

Abstract

Procedural content generation (PCG) is a powerful and convenient tool that is used to algorithmically generate content instead of handcrafting it. Widely used in video game development, procedural content generation automates certain parts of the development process that can otherwise be time-consuming, such as creating game levels and building terrain. Roguelike games are a video game genre known for having procedurally generated dungeons. In this thesis, we have created a procedural content generator for two-dimensional roguelike dungeons and evaluated its performance. We present two different PCG algorithms, one built using breadth-first search and one using the concepts of depth-first search with backtracking. Additionally, we then use the generated layout to create a three-dimensional representation of the dungeons using the Unity game engine. We attempt to evaluate the "enjoyability" of the generated dungeons objectively, by creating a new metric called the "Choice heuristic" and modifying an existing metric that Gellel and Sweetser proposed. Based on the information from our evaluation, we conclude that the generator using the breadth-first algorithm generates dungeons that are more enjoyable to explore compared to the generator using the depth-first algorithm.

Acknowledgments

I would like to acknowledge my advisor, Professor Daniel Palmer, as the completion of this IS would not be possible without his feedback, advice, and words of encouragement. I would also like to thank my friends for being supportive throughout my time on campus.

Contents

List of Figures

CHAPTER 1

Introduction

Over the years the video game industry has grown exponentially. In 2021, The size of the global video games market was valued at USD 188.73 billion, and it is projected to grow to USD 307.19 billion by 2029 [9]. The growth of the video game market can also be attributed to the lockdown caused due to COVID-19 as most people were using video games to keep themselves busy while they were stuck at home. Video games are a form of entertainment which similar to movies, are made from the combination of various components such as game objects (props), scripts (code and algorithms), and audio (soundtracks and dialogues). We call these different components that make up a video game as content. The content of most video games is created by various groups of people such as audio engineers, voice actors, game designers, and so on. This makes video game development time-consuming and expensive. The complexity of the development process can lead to the creation of video games that are rushed, incomplete, and littered with bugs. Furthermore, there has also been a growing concern for video game developers as it has been reported that about 50% of developers (mostly freelance developers) are expected to work for long hours consistently or experience a crunch period during which they are required to work overtime [1].

Procedural content generation is the process of using algorithms or systems that algorithmically generate content based on the indirect input or constraints set by the developer. It is a convenient and powerful tool that has been used to alleviate some of the burdens for video game developers. It is able to create endless content, as per the requirement of the developer. Procedural content generation is similar to the automation of certain operations within a factory. We are making an algorithm to do work that would otherwise be time-consuming and expensive. For example, if we want to create a video game that has multiple cities, we can speed up the process by creating a procedural generator that can create cities for us instead of manually modeling the city. The generator will create a random city based on inputs given by us such as the population of the city, and the number of houses. Other than being able to speed up the development process, procedural content generation can be used to inspire developers and implement systems that can add replay value to video games. Outside of video games, procedural content generation is commonly seen in areas such as art, simulations, and AI systems.

The usefulness of procedural content generation in video games can differ based on the genre of the video game. For example, video games that are aiming to provide a cinematic experience revolving around a fixed plot such as The Last of Us, will not benefit significantly from it. This is because video games such as these already have a fixed narrative through which the content revolves around. Procedural content generation can be used, but the content produced by the generator will need to be heavily modified as a generator will tend to create content that is more generalized. On the other hand, sandbox video games such as Minecraft would benefit greatly from procedural content generation. Sandbox video games are a genre where the video game provides a large, open playground where the players can interact with the world using the tools and gameplay mechanics provided. If the world of Minecraft were the same every time a player started over, it would lead to a boring experience with no replay value. This is why some sandbox video games such as Minecraft and Terraria have a procedural content generator that randomly creates a world each time a player starts a new world. These examples also showcase that procedural content generation can be used to assist game development or be integrated within a video game where it is able to create content in run-time or real-time.

Another popular genre of video games known for having procedurally generated dungeons is roguelike games. “Roguelike" is a subgenre of the popular genre, Role Playing Games (RPGs). RPGs are a genre of video games where the player assumes the role of a character within a fictional world. Roguelike games are named after the video game Rogue, which is known for having procedurally generated levels, permanent player death, and a challenging gameplay loop [13]. Other video games have tried to replicate and build upon this formula which subsequently led to the birth of this genre. Roguelike games are usually simple in nature. For example, older roguelike games would have two-dimensional dungeons which the player can explore. The dungeon’s layout and the contents of its rooms are procedurally generated and usually represented in text [13]. Players would simply roam around and explore these dungeons. The goal of this IS project is to build a procedural content generator that can create a two-dimensional layout of a dungeon as seen in a traditional roguelike game. Furthermore, the goal is also to be able to use the generated layout to create a three-dimensional dungeon using the Unity game engine.

In the second chapter of this thesis, the roguelike genre of video games will be explored. Information regarding its origin, its key characteristic, and its relationship with procedural content generation will be covered. In the third chapter of this thesis, the concept of procedural content generation will be discussed in detail. Including: its application in video games and its properties. The fourth chapter will cover two different approaches to procedural content generation in games that are relevant to this thesis. In the fifth chapter of this thesis, the existing research on the different methods to build a procedural content generator for roguelike dungeons will be discussed. In the sixth chapter of this thesis, the tools and algorithms that will be used to build a procedural dungeon generator will be covered. In the seventh chapter of this thesis, the procedural dungeon generator I built alongside its application in generating a three-dimensional dungeon in Unity will be discussed. The underlying concepts, tools, and algorithms that I used to create my procedural content generator will be covered in great depth. In the eighth chapter of this thesis, the procedural content generator I created will be evaluated. Finally, the ninth chapter will be the conclusions I made based on the research I conducted throughout my thesis.

CHAPTER 2

Roguelike Games

Role-playing games, commonly abbreviated as (RPGs) are a genre of video games where the player takes on the role of a character in a fictional or historical setting. Roguelike games are a subgenre of RPGs alongside other popular subgenres such as massively multiplayer online RPGs (MMORPGs) and Japanese RPGs (JRPGs). Roguelike games have grown popular due to their contrasting differences from the other subgenres within RPGs. Most RPGs are characterized by work-intensive developer-designed levels, a linear story that does not change with each play-through, and forgiving design elements such as checkpoints, which allow the player to continue when they fail an objective. Roguelikes, however, are characterized by features such as procedurally generated levels, permanent player death, management of finite resources to survive, and the requirement of players to be skilled and lucky due to the levels being different each time they play. While these elements are commonly seen in roguelikes, these are not strict requirements. There are roguelike games such as Mystery Dungeon which combine popular elements from JRPGs such as turn-based combat. As time has passed, there have been many games that have branched off the original idea of roguelike games and expanded upon these core features [13].

Procedural content generation is a key element that defines roguelike games. It allows roguelike games to become simple and replayable. The simplicity of the roguelike formula and the efficiency of procedural content generation have attracted many developers with small teams and small budgets, known as indie developers. These developers have created some popular indie roguelike games such as Hades, The Binding of Isaac, and Risk of Rain 2, which have all been highly praised and awarded.

2.1 Origin of Roguelike Games

This genre of games was popularized by Rogue, a dungeon-crawler video game officially released in 1984. Rogue is a video game first developed in 1980 by Glenn Wichman and Michael Toy, who at the time were students at UC Santa Cruz. Their vision was to make a video game with a randomized maze layout and randomized monster/item locations. They also wanted to incorporate elements from Dungeon and Dragons. The developers simply distributed their game across campus, but the game was officially released in 1984 [13].

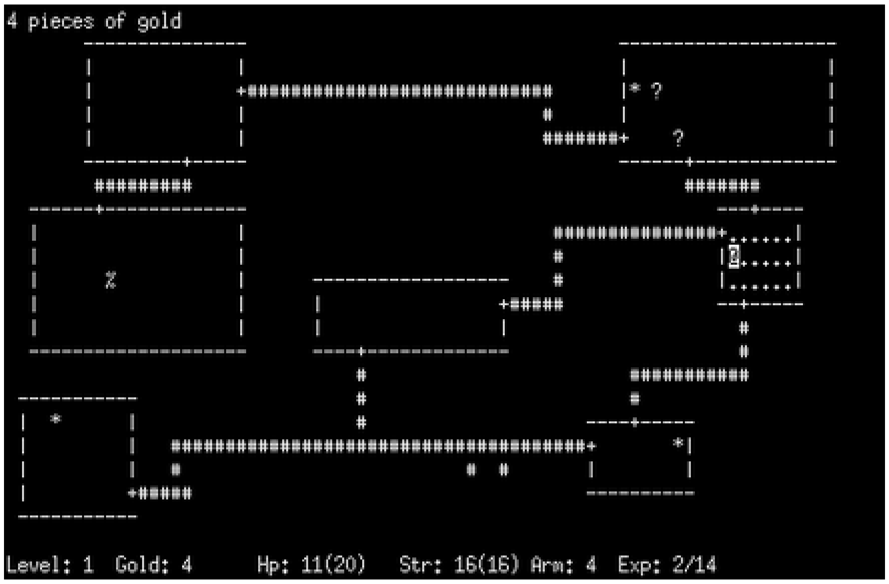
In Rogue, the player controls a character through a dungeon with multiple levels. Unlike other RPGs, the character cannot continue their progress once they die. This design element is called permadeath. This forces the player to start a new game every time they play. However, the dungeons are procedurally generated, making each playthrough unique.

The game's popularity inspired many other games such as Hack, Nethack and Moria. The name "roguelike" came from discussions surrounding these games in 1993. As all these games are similar in nature, people wanted to come up with an umbrella term to facilitate discussions around these types of games. After debates among users, the term "roguelike" was picked as Rogue was the oldest of these types of games [25].

2.2 Roguelike Dungeons

Dungeons are typically represented as an enclosed area that consists of rooms that are connected by hallways. The player explores the dungeon while completing various objectives such as looting treasures and fighting monsters. While dungeons encourage player exploration, there are tight constraints when it comes to the progression of objectives. This is to ensure that the dungeon can be completed and will encounter a situation they cannot complete. In video games, this is called a "softlock" [15].

The dungeons of older video games such as Rogue have simple dungeon layouts. The dungeon consists of square or rectangular rooms with pathways that connect these rooms. Items and enemies are scattered across these rooms. Since older computers ran on a command line interface, the dungeon would be a two-dimensional layout represented in text using symbols such as "#", "-" and "+". This form of graphical representation is now referred to as "ASCII art". An example of a dungeon from Rogue can be seen below.



```
4 pieces of gold
```

```
Level: 1  Gold: 4      Hp: 11(20)    Str: 16(16) Arm: 4  Exp: 2/14
```

Figure 2.1: Example Dungeon from Rogue (1980 Version) [22]

Newer video games now have a large variation when it comes to representing a dungeon. Developers are now able to make three-dimensional video games and dungeons with various different shapes with geometric complexity.

This thesis will be focusing on procedurally generating a simple two-dimensional layout of a dungeon using ASCII art. The dungeon will have a similar structure to the dungeon seen in Rogue. The information from the two-dimensional layout will be used to create a three-dimensional layout.

CHAPTER 3

Procedural Content Generation

Procedural content generation (PCG) is the algorithmic generation of content with limited or indirect user input. Content in this context refers to the pieces that make a product. For example, content in video games includes levels, characters, items, quests, etc. It is a convenient tool that allows us to create an endless stream of content that suits our needs, without having to build it from the ground up. In this chapter, we will be going over why procedural content generation is pivotal in video games. Furthermore, we will also be discussing the important properties of procedural content generators and the different types of procedural content generators.

3.1 Procedural Content Generation in Video Games

While procedural content generation has a wide variety of uses in fields such as music, art, simulations, and AI systems, most of its use and research have been focused on its importance in video games and video game development. Attempts at utilizing procedural content generation in video games have a long history. In 1980, the video game Rogue pioneered procedural content generation by procedurally generating its dungeons. In the modern context, procedural content generation is almost always used during the development of a game [28]. The use of procedural content generation in video games and its importance revolves around 4 key factors. They are efficiency, diversity, creativity, and replayability.

3.1.1 Efficiency

Video game development requires a lot of time and effort. A large part of the development costs is taken by the creation of game content. While we might expect the development process to become easier with the rapid advancement of technology, it is also important to keep in mind that video games have also become increasingly more complex [28]. For example, Super Mario Odyssey released in 2017, is miles more complex than Super Mario Bros., released in 1985. The creation of content is why procedural content generation is used to alleviate some of the burdens by automating certain aspects of it. Furthermore, it also allows for smaller game development companies to work on large projects without a sizeable impact on the quality of the product.

3.1.2 Diversity

Procedural content generation adds depth to a video game's experience by creating novelty and variety. For example, procedural content generation can create unique and varied environments or levels that players can explore. Furthermore, PCG of other forms of content such as items, quests, and characters can also be combined to allow for an endless possible combination. There are also PCG systems that combine PCG with neural networks in order to generate content that is tailored to the tastes of the player. An example of an implementation of this can be seen in the video game Left 4 Dead 2. In this game, a PCG system is used to dynamically adjust the game's difficulty based on the player's performance [12].

3.1.3 Creativity

Procedural content generation can allow developers to become more creative. This is because PCG systems can produce radically different content compared to something that a human would make [15] This can inspire developers and help them think outside of the box. Furthermore, we can also say that PCG allows the creativity of some developers to be realized. For example, No Man's Sky is a video game where the player explores a vast and diverse universe with billions of planets, that is procedurally generated. Without PCG, video games such as No Man's Sky would not have been possible.

3.1.4 Replayability

Replayability refers to how likely a person is willing to play a video game repeatedly. Replayability is important for video game developers because they want to be able to create an experience that a player deems worthy of experiencing again. Earlier we discussed how procedural content generation adds diversity to a video game by creating a different and unique experience. We also mentioned No Man's Sky, which is a video game where a player can explore a universe similar to the size of our universe, that is procedurally generated. From the examples and benefits of PCG that have been mentioned so far, it is clear that video games utilizing PCG become more replayable. Furthermore, the replayability of a video game also adds to its longevity and support. Longevity in this context refers to how active and alive the community of the video game is. If people enjoy and replay the game, they have more of a reason to support the game and its developers. From there, the developers also gain a financial incentive as they can continue to support the game or work on a sequel that the players will be anticipating.

3.2 Application of PCG in Video Games

In this section we will briefly discuss the application of procedural content generation in three popular video games.

3.2.1 Minecraft

Minecraft is a sandbox game that has a world that is procedurally generated. The terrain of Minecraft, like most 3D video games, is entirely based on noise functions. When you begin the game, a 64-bit number called a seed is pseudo-randomly selected [5]. The map seed is pseudo-random as it is obtained from the system clock of the computer at the time the world is created [24]. The map seed can also be given by the player manually. The seed is used to generate noise functions. The output of these functions determines the terrain's characteristics such as its height and where biomes are located [5]. Biomes are different types of terrains such as deserts, rainforests, and snowfields. The procedural generation of the world is deterministic as we can replicate the results of the generator by making use of the same seed. The procedurally generated world can be infinitely large. However, there is a limit on the size of the vertical plane, but the horizontal plane can be infinitely large.

3.2.2 No Man's Sky

No Man's Sky is a video game that features a procedurally generated universe with over 18 quintillion (1.8×1019) planets to explore [23]. Similar to the procedural content generator used in Minecraft, the generators of this game also use a 64-bit seed number. However, unlike Minecraft, the seed cannot be provided by the user, and the entire universe of the video game revolves around one single seed called the founding seed. The founding seed is the phone number of one of the developers of the game. Using the seed mentioned previously, the first generator creates the universe, the positions of the stars, and the type of the stars. The second generator uses a pseudo-random seed based on the position of the star to define its planetary system. Finally, another generator uses the position of a planet as a seed to define its planetary features. There are many more procedural content generators within this hierarchy of generators that define other smaller features such as the plants and items found within the planets [23]. The generators are also deterministic, which allows No Man's Sky to become a multiplayer video game where each player would be within the same universe. Despite the scale of the universe, it does not take up a large amount of space. This is because only the sections of the universe that are within proximity to the player are loaded and rendered.

3.2.3 Left 4 Dead 2

Left 4 Dead 2 is a first-person shooter game where the player needs to fight a horde of zombies to progress through a linear story. The game has a PCG system called the AI director which combines neural networks and PCG to alter the game dynamically in real-time based on the player's experience. The AI director can spawn hordes of zombies and boss enemies. It can also change the placement of weapons and health items and modify certain parts of the level. For example, if the player is low on health, the AI director can increase the probability of finding a medical kit in place of a pain killer, which is normally found inside a medical cabinet [21].

3.3 Properties of a Procedural Content Generator

In this section, we will discuss the important properties that define procedural content generators. Depending on the use case of the generators, some of these properties are more favorable than others. The 4 important properties of a procedural content generator are:

- Speed – In most cases, a procedural content generator should be able to create a large amount of high-quality content quickly and efficiently. Depending on the use case of the generator, the speed of the generator could vary [15]. For example, The AI director in Left 4 Dead 2 would require to be fast as it is responsible for altering content during gameplay. On the other hand, a generator that is being used during the development of a game would not be required to be as fast.

- Reliability – A procedural content generator is reliable if it can accomplish the task it is required to do [15]. For example, the world generator of Minecraft must be reliable as its failure will negatively affect the player's experience. However, a generator used during video game development is not required to be perfect.

- Controllability – A procedural content generator is controllable if it provides the user with some control over the outcome of the content generation process [15]. A generator can be controllable by allowing the user to specify and change constraints. Controllability can also be added by making the generator deterministic. A deterministic generator allows the user to replicate previous results by using the same variables. Finally, Controllability can also allow for scalable and versatile generators such as the one that is seen in No Man's Sky.

- Expressivity and diversity – A diverse and expressive procedural content generator offers a wide range of possibilities, and it produces content that is unique and not repetitive. Diversity can be added to a generator by adding randomness into the content generation process. In some cases, a generator is too expressive, which leads to the generated content being senseless in the context of what is required [15].

3.4 Taxonomy of Procedural Content Generation

3.4.1 Online Generators

Online generators produce content in real-time while the player is playing the game. This allows for the generation of player-adapted content which can enhance the player's experience. Online generators, are required to be fast and adaptive while providing consistent content with quality. Online generators can be used when the designers want procedural content generation to have a dynamic role in the video game where it can influence gameplay itself. Online generators typically make use of scaling or evolutionary algorithms as it allows the content to adapt and expand over time and keep track of player preferences [15]. The AI director used within Left 4 Dead 2 is a great example of an online generator.

3.4.2 Offline Generators

In most cases, offline generators are static, and applied during game development or while the game is loading. During game development, offline generators can aid in prototyping and design by giving developers the flexibility to experiment while imposing specific constraints for content. This can make the development process more efficient and speed up production time. For example, the video game The Elder Scrolls IV: Oblivion by Bethesda Softworks used procedurally assisted methods to create most of the terrain and forests. Furthermore, popular game engines such as Unity also allow the user to procedurally generate terrain using a mesh. In a video game, an offline procedural content generator could be used to create and populate a game world [15].

3.4.3 Deterministic and Stochastic Generators

A deterministic generator allows us to recreate the same content given the same conditions and parameters. While a stochastic generator does not allow for the recreation of the same content even if the conditions and parameters are the same [15]. The world generator of Minecraft is a great example of a deterministic generator.

CHAPTER 4

Approaches to Procedural Content Generation in Video Games

There are many approaches to procedural content generation in video games. This section focus will focus on the two major approaches that are relevant to this thesis. They are the search-based approach and the constructive approach. We will discuss their methodology and applications.

4.1 Search-Based Approach

The basic principle behind using a search-based approach is the idea that we can reach the desired solutions as long as we find partial solutions to the problem and tweak them by keeping the properties we deem beneficial, and discarding the properties we deem harmful. A search-based approach typically has content representation that represents the properties of the content that we want to create. Using the content representation, a large population of content is randomly generated. The search algorithm is the core of this approach which is typically a stochastic search algorithm such as the evolutionary algorithm. The search algorithm is used to search through the large population of content in order to find content with the desired qualities. The content is then evaluated by a fitness function and based on its result; the next generation of the population is reproduced. We will be discussing these core components of the search-based algorithm by looking into the most popular searched based algorithm, which is the evolutionary search algorithm.

4.1.1 Evolutionary Search Algorithm

An evolutionary algorithm is a stochastic search algorithm that is inspired by the biological concept of evolution. They are a heuristic-based approach to solving problems that are complex in nature. An evolutionary algorithm solves a problem by first creating a population of possible solutions to the problem which is then evaluated by a fitness function that indicates how good the solutions are. The solutions that have the highest scores are now used to evolve the population in order to create solutions that score better than the previous generation [11].

4.1.1.1 Content Representation

A central question when making use of evolutionary algorithms is how we are going to represent the content that is going to be evolved. Finding a solid answer to this question is important because having well-defined properties allows for a thorough evaluation of our content. In simple words, content representation is the definition of the properties in our content. Like genetics, the properties of our content can be divided into phenotypes and genotypes. Genotypes are the set of genes that an organism carries, in this context it is the set of properties and rules that defines the content. Phenotype, on the other hand, is the physically observable characteristics of the organism that is influenced by its genotypes, which in this context would be the characteristics of the generated content. For example, if we are thinking of creating a dungeon for a video game, certain properties such as the number of rooms and the position of enemies and items would need to be specified. Using this information, multiple dungeons can be created. While all of the dungeons will have the same genotype, there is a high probability that they will look different.

For example, there can be a 10-room dungeon with no branching pathways and there can also be a 10-room dungeon with many branching pathways. Content representation can be done through the use of variables, graphs, and data structures such as objects and arrays [15].

4.1.1.2 Fitness Function

Once the content representation is defined and the content is generated, it needs to be evaluated. This is done through the use of fitness functions which assigns a score to each generated content in the population either in the form of a scalar or a vector. The score reflects how desirable the properties of the content are in the given context. To create an effective fitness function, the developers need to understand what properties need to be optimized. For example, one important property could be enjoyment. The issue with such a property is the lack of a proper definition and its abstract nature. Developers need to break down these broad properties into multiple measurable properties.

4.1.2 Application of the Search-Based Approach

Togelius et al. evolved racing tracks based on the players' playstyle. The game is a simple 2D racing game. In this game, the tracks were represented as vectors of real numbers. The numbers in the vectors are control points for b-splines, which are a sequence of Bezier curves [15].

First, a model that represents a player's playstyle is created. This is done by teaching a neural network to drive like the player. Once the model is created, a generated track is evaluated by simulating how the real player would perform on the track using the trained neural network model of the player. Furthermore, the information from this simulation is used by three different evaluation functions that measure if the track is challenging and diverse. The algorithm made use of cascading elitism, which is an algorithm that has several stages of selection to apply a necessary selection pressure on all the optimization objectives [15].

A search-based approach is beneficial when the content we are generating does not have a strict structure. If the developer is not sure about what is desired, the search-based algorithm will find the desired solution as long as some basic constraints are given. Since it generates the content and evaluates them later, the search-based approach is not efficient as the constructive approach, which is the approach we will be looking into next.

4.2 Constructive Approach

If the developer has a clear understanding of what they want to generate, it is more efficient to skip the evaluation process. Since certain outcomes of the generator are predetermined by the constraints given, the constructive approach allows for greater control of the outcome of the generator. Similar to the search-based approach, the constructive approach also requires a well-defined content representation [15].

A common example that uses the constructive approach is the generation of dungeons in roguelike games, which is the focus of this research. A dungeon can be complex or simple, but its structure is rigid. Constructive procedural generators are also fast allowing them to be a solid choice in the creation of online procedural content generators [15].

4.3 Procedural Generation of Dungeons

While dungeons can be created without procedural content generation, genres of video games such as roguelike games make use of procedural content generation.

The constructive approach is commonly used in the construction of dungeons and levels within video games [15].

To generate such a dungeon using the constructive approach, a content representation that describes the rules of the game is required alongside an algorithm that can create the dungeon from said content representation.

4.3.1 Spatial Representation using Cellular Automatons

Before we can generate the dungeon, we need to represent the dungeon within a virtual space. In the generation of a simple 2D dungeon, data structures such as arrays, linked lists, and trees can be utilized. In this case, each index or node will represent a room. Another popular method of representing a dungeon is using cellular automata.

A cellular automaton is a discrete-time model that provides spatial representation in the form of a uniform grid. A cellular automaton consists of an n-dimensional grid with a set of rules and transition rules. Most cellular automata are 1 dimensional or 2 dimensional. Each cell in the automaton has an initial state which evolves based on the rules of said automaton. At every time interval t, each cell decides what its new state will be based on the state of itself and all the cells in its neighborhood in the previous time frame $t - 1$. In a 2D cellular automaton, the neighborhoods are either Moore neighborhoods or von Neumann neighborhoods. Both neighborhoods can have a whole number of sizes one or greater. A Moore neighborhood is a square. A Moore neighborhood of size 1 consists of the eight cells that immediately surround a cell, meaning it also includes the cells diagonal to it. A von Neumann neighborhood is like a cross or a plus (+) symbol. A von Neumann neighborhood of size 1 will consist of the four cells that immediately surround a cell. The four cells are the cells to the right, left, top, and bottom of the cell. The number of possible configurations of the neighborhood is dependent on the number of states for a cell to the power of the number of cells in the neighborhood [15].

There are many ways to utilize a cellular automaton to represent a dungeon. In the first method, we can assume that a cell within a cellular automaton represents a room. This method is great for simplifying the representations of rooms. In the second method, each cell within a cellular automaton can be assumed to be a building block or a Lego that builds certain sections of a dungeon. For example, one cell can be considered a wall block and another cell can be considered a floor block. In order to create a room within a dungeon, we would need multiple wall and floor blocks. This allows for the creation of more complex dungeons with varying room sizes and pathways. Now that the space of the dungeon has been represented. An algorithm that can use the content representation provided by the cellular automata will be needed in order to build the dungeon.

4.3.2 Application of the Constructive Approach

Johnson et al. were able to describe a procedural content generator for generating an infinite dungeon using cellular automata. The cellular automata in this research use 5, 50 by 50 square grids. The initial grid is called the central base and then four adjacent grids are built to the north, south, east, and west of the central base grid. Each cell within the grid contains information about the cell's location in the grid, the state of its neighbor cells represented by an aggregated neighborhood value, the type of the cell (floor, wall, or rock), and the cell's group number. The neighborhood value is the sum of rock cells available within the neighborhood. The floor and rock combination are used to create a tunnel-like cave [10].

The cells of the central base grid are initialized with floor cells. The algorithm uses uniformly distributed random numbers to convert 50% of the floor cells into a rock cell. Now the neighborhood value is calculated by using a Moore neighborhood of value 1. If the neighborhood value is greater than or equal to 5, a cell becomes a rock cell, else the cell becomes a floor cell. The cellular automata use an iterative algorithm to run this calculation as many times as the user specifies. The higher the number of iterations, the wider the cave (more floor, less rock) [10].

The cells of the dungeon of the game could stretch out infinitely and the content is generated in real-time while the game is running. The game renders one room at a time and when exiting the room, the game has a time window that allows it to generate the new room that the player will be entering [15].

CHAPTER 5

Previous Research

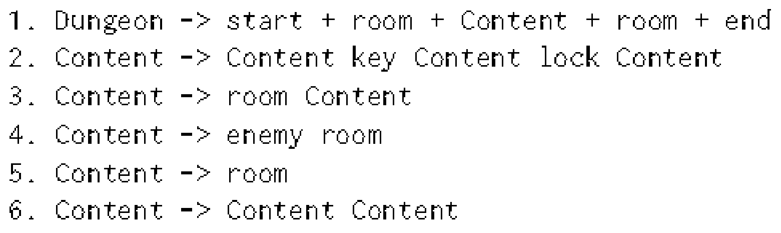
This chapter will go over the previous research that has been conducted on the topic of creating a procedural dungeon generator. The two research papers covered in this chapter inspired the methodology used to build the procedural dungeon generator in this thesis.

5.1 A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels

The authors of this research paper are Alexander Gellel and Penny Sweetser, and it was published by the Association for Computing Machinery (ACM) in 2020. At the time this paper was written, both authors were a part of the Australian National University. Sweetser in particular is currently a senior lecturer for the Australian National University and leads research in areas such as AI, player experience and human-AI interaction in games [19]. She has also worked as a game designer for companies such as 2K games and Creative Assembly [17]. Since this is a summary of their research, in-text citations will not be provided.

In their research, Gellel and Sweetser explored various approaches to procedural content generation in video games in order to create their own prototype procedural content generator. After their analysis, they aimed to create a prototype procedural content generator by hybridizing two approaches to procedural content generation. The first one being generative grammars (context-free grammar) and the second one being cellular automata. The dungeon will be represented as a grid similar to video games like The Legend of Zelda, where the cells of the grid are rooms that are equally sized. The dungeon created by Gellel and Sweetser makes use of lock-key pairs to test the player's spatial awareness and navigational memory.

To define the constraints and rules of their dungeon, Gellel and Sweetser made use of generative grammars. The grammar rules are used to recursively generate a single string which is used to describe a flow and intended order of room types that a player will visit. Each generation begins using a rule defining the start and the endpoint. The content of the dungeon is filled recursively. The rules are made strict to force all the keys in the dungeon to be generated with appropriate pairs of locks. Furthermore, the grammars also ensure that a key always appears before a lock. Below in fig. 5.1, is an example of a simple recursive grammar provided by Gellel and Sweetser.



```
1. Dungeon -> start + room + Content + room + end
2. Content -> Content key Content lock Content
3. Content -> room Content
4. Content -> enemy room
5. Content -> room
6. Content -> Content Content
```

Figure 5.1: Example of a Simple Recursive Grammar

To represent and generate the space of the dungeon, Gellel and Sweetser used an approach that is inspired by the behavior of cellular automata. The approach is considered inspired because it is not a traditional cellular automaton as it makes use of non-determinism and information beyond a cell's immediate neighbors is provided. Since the dungeon will be represented as a grid, they can be modeled as cells of the cellular automaton without additional changes. The generator is given a single string which is generated by the grammars that were described earlier. It uses this string to generate the dungeon with the correct order and placement of the lock and keys and the type of rooms.

The generator has two cellular automaton rules or states. In the first rule, the generator is given the location of its starting room and from there, it pseudo-randomly selects a cell from its immediate neighborhood that is not a room. The generator moves to this cell and places a room, and it is assigned a type based on the mission string. In this rule, the generator does not maintain knowledge beyond the neighborhood of the current cell. This rule is repeated until it reaches the end room or until it is incapable of making a move. In the second rule, also called the Subsection Search, the generator attains knowledge beyond the neighborhood of the current cell. In this rule, for any given subsection of the map, the generator evaluates its surroundings and picks the first random valid direction branching off from the room that is closest to that section's average center point as it can reach, based on Manhattan distance.

Using those two rules, Gellel and Sweetser created two search methods/algorithms that are used by their generators to build a dungeon. The first search method is called the Persistent Subsection Search. In this search method, the generator will always make use of the second rule to generate the dungeon. The second search method is called the Subsection Search on Halt. In this search method, the first rule will be used to generate the dungeon until a dead end is encountered. Once a dead end is encountered, it will make use of subsection search, which is the second rule. One extra rule is added such that the generator can overwrite its current location if no possible moves exist even after employing the second rule. This third rule forces an end room to be placed even if an unimportant room must be removed for this to occur. Gellel and Sweetser mention that it is equivalent to repairing and it acts as a failsafe that ensures that the dungeon is complete and playable.

The generator is assembled in two parts. The first part is the generator that creates the input string using a generative grammar. It is implemented by a Python script. The space generator using the cellular automaton inspired behavior, was implemented using the Unity Game Engine. Below is an example of a dungeon provided by Gellel and Sweetser. The dungeon in fig. 5.2 was generated using persistent subsection search and the dungeon in fig. 5.3 was generated using subsection search on halt.

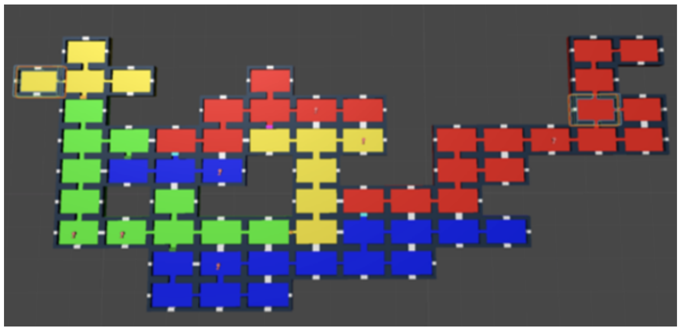


Figure 5.2: Dungeon Generated Using Persistent Subsection Search

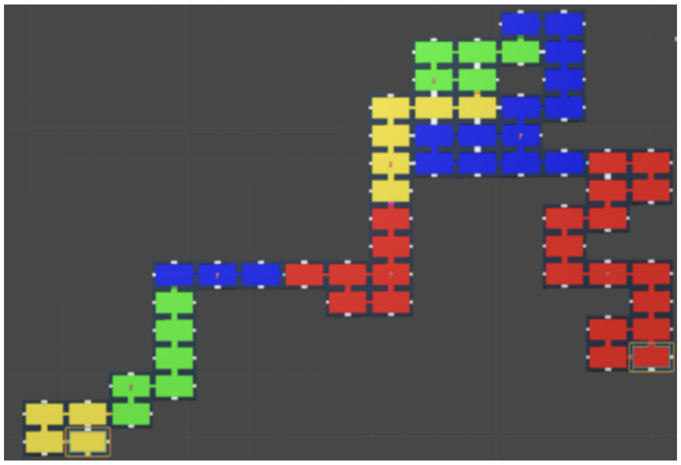


Figure 5.3: Dungeon Generated Using Subsection Search on Halt

After Gellel and Sweetser created their generators, they evaluated the generators and the dungeons generated by them. The generators were assessed based on their ability to produce levels that are complete and playable and their speed. Gellel and Sweetser assessed the generated dungeons by creating a metric called the path difference heuristic. The heuristic was calculated by taking the difference of the critical path and the spine. The critical path is the length of the path required to reach the final room by collecting all the keys. The spine is the shortest path required to reach the final room without the keys. They argued that this heuristic allows them to capture the "interestingness" of a dungeon as a significant difference would mean that the player is required to explore more before they can reach the final room.

Gellel and Sweetser concluded that a generator using only the first rule was not able to generate complete and playable levels consistently. The generator using the persistent subsection search generated dungeons with a more varied distribution of rooms with fewer long uninteresting pathways compared to the generator using the subsection search on halt. This is also reflected in the path difference heuristic scores in which the persistent subsection search scored higher. Generators using the persistent subsection search and the subsection on halt search were able to make dungeons that were complete and playable 100% of the time.

This paper inspired most of the key elements of the dungeon that was created in this thesis. The first thing it inspired was the spatial representation of the dungeon and its basic characteristics. The use of a grid to represent the space of a dungeon is intuitive and as Gellel and Sweetser says, it is simple and easy to define. The depth-first search algorithm and the breadth-first search algorithm I will be using to create my generator are almost identical to the rules and search methods used by the generators created by Gellel and Sweetser. The paper also inspired me to create a three-dimensional representation of the dungeon in Unity and divide the generator into two parts. The first part being a generator implemented in Java, that generates a two-dimensional dungeon and the second part being a generator implemented in

Unity, that creates a three-dimensional representation of the dungeon, by utilizing the information obtained from the previous generator.

5.2 Generating Procedural Dungeons Using Machine Learning Methods

The authors of this research paper are Mariana Werneck and Esteban W. G. Clua. It was published by the Institute of Electrical and Electronics Engineers (IEEE) in 2020. At the time this paper was written, both authors were members of the department of computer science of the Fluminense Federal University (Universidade Federal Fluminense) in Brazil. Clua in particular is currently a professor at the university and leads research in areas such as digital games, virtual reality, GPUs, and visualization. He is also the coordinator of the NVIDIA Center of Excellence which is located at the CS institute of the university [27]. Since this is a summary of their research, in-text citations will not be provided.

After looking into previous attempts to create a procedural content generator for roguelike dungeons, Werneck and Clua created their own procedural content generator using machine learning methods. The method proposed uses machine learning agents to first create the rooms and path and decide on the placement of these rooms. The proposed method is divided into two big phases. The first phase revolves around the design of the dungeon's general structure and the second phase around the training and implementation of the machine learning agents.

The first thing that was considered in the design are the type of rooms and their relative configuration. Werneck and Clua argue that specific rooms with events such as a boss fight, and equipment acquisition via a treasure or a shop give the player a sense of progression in the dungeon. They argue that these rooms with important events need to be placed strategically. The event rooms chosen for their paper were the boss room, the treasure room, and the shop room.

The next thing that was considered is the spatial representation of the dungeon. To represent the space of their dungeon, Werneck and Clua are using a square matrix. This space is the same as a grid. The size of the grid depends on how big the dungeon is intended to be. The starting room is randomly chosen but it cannot be chosen in the border of the matrix because it prevents the starting room from having more neighboring rooms. After this, random positions are chosen to assign the event rooms. There is a design rule when it comes to selecting these event rooms.

Once the spatial representation of the dungeon was created, Werneck and Clua shifted their focus toward training and implementing their dungeon-generating agents. They are defined as intelligent agents that are responsible for creating a path to a specific event room and choosing where the event room will be placed. The agents will be specialized such that an agent is only trained to create the path and placement of only one event room. This would mean that to generate a dungeon they would need multiple agents. The agents were trained using the reinforcement learning method available in the Unity Machine Learning Agents Toolkit. A square matrix was used to represent the dungeon space in the training environment and every matrix cell represented a possible position for a room. The action an agent can take is navigating up, down, left, or right of the matrix and stopping to create a path. The agent will place a room where it stopped creating a path and the room is assigned the type based on what event room the agent is specialized in.

Once the agents are trained, the last step is to represent the square matrix dungeon generated by the agents into a three-dimensional game dungeon. The prefabs of the rooms are premade, and the rooms will be rendered in the positions that are marked on the matrix. The doors connecting to the rooms are positioned at every intersection between two rooms. This means that there will always be a door between two adjacent rooms. Below in fig. 5.4 is an example of a completed dungeon made by the trained agents. The room in red is the boss room, the room in light blue is the start room, the room in green is the shop room and finally, the room in yellow is the treasure room.

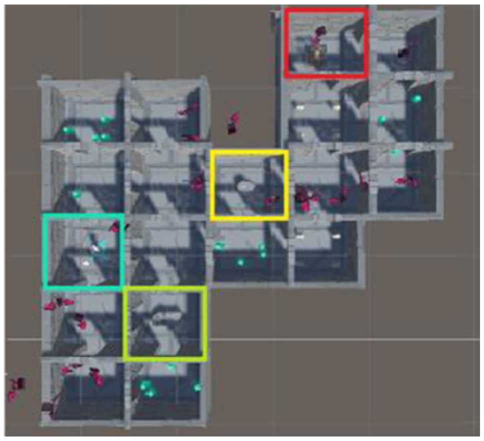


Figure 5.4: Dungeon Generated Using Machine Learning Agents

To evaluate the generated dungeons, Werneck and Clua conducted a survey with 15 users. In the survey, users were asked to play a simple game where the objective was to find a princess and free her. The player had to explore the dungeon and beat the boss to free the princess. Players were asked to play two versions of the game, one implemented using the agent dungeon generator and the other which was manually made. The survey results show that the dungeons generated by the agent generator were rated as more enjoyable with better maps and better replayability. However, the participants could guess which of the dungeons were procedurally made because they were more challenging than the manual dungeons.

This paper helped reinforce some of the ideas I had come up with for my procedural content generator. The first one is the use of a square matrix to represent the space of the dungeon. It is almost identical to the grid representation that I will be using for my procedural generator. Furthermore, this paper inspired me to create different room types such as the treasure room and the enemy room. While I had already planned to make a three-dimensional representation of the generated dungeons, the examples of the completed dungeons given in this paper inspired how my representation will look like.

CHAPTER 6

Relevant Algorithms

This section will be going over the concepts and algorithms that were used to create my procedural dungeon generator. Understanding these concepts can aid in the understanding of the upcoming chapters.

6.1 Graphs and Trees

A graph is a non-linear data structure which represents a set of objects called nodes (or vertices) and the connections between them called edges (or arcs) [8]. They are commonly used to model the relationship between objects such as computer networks or a family tree. An example of a vertex is a node, which is commonly seen in a tree and a linked list, which are also a type of graph. Edges are used to connect two vertices in a graph.

Trees have a hierarchical structure, where each node has a parent, and zero or more children, except for the root node, which is the topmost node in the tree [8]. The leaf nodes or terminal nodes are nodes that do not have any children. There are algorithms that allow us to search through a tree. The two search algorithms relevant to this thesis are depth-first search and breadth-first search.

Graphs and trees are important in this thesis because we assume that the structure of a dungeon is similar to that of a tree. Each node in the dungeon will be a room, and each edge will create a doorway that connects these rooms. Furthermore, the root node is the starting room, the branches of the trees are branching pathways, and the leaf nodes are dead-end rooms. The final room or destination room is the final terminal node that is generated. With this assumption, we can use the concepts of the search algorithms highlighted above to procedurally generate a dungeon.

6.2 Depth-First Search

Depth-first search is a graph traversal algorithm that visits all the nodes in the graph by exploring as deeply as possible along each branch before backtracking [3]. Depth-first search can be implemented recursively or by using a stack. The recursive implementation starts at a node and visits its neighbors recursively until all the nodes have been visited. The stack implementation keeps track of the vertices that it has visited and pops the stack to visit its neighbors. If it reaches a terminal node, it will use the stack to backtrack to a node to begin visiting the nodes of another branch [8]. The time complexity of the depth-first search algorithm is O (N + E) where N is the number of nodes and E is the number of edges in the graph. This is because this algorithm visits every node and edge exactly once [8].

If we traverse through a tree using the stack implementation for example, the depth-first search algorithm starts its exploration from the root node. From there it will keep track of the nodes that it has visited. The root node will be considered visited and it will be popped from the stack. The adjacent, child nodes will be added to the stack and one of them will be popped based on the rules of the algorithm. Once the node is popped from the stack, its child nodes will be pushed on the stack. The cycle continues until it pops the terminal node of the branch it is exploring. With the stack's help, the algorithm can backtrack to the previous adjacent node and continue its search in another branch.

Now let us apply this traversal to the example tree below in fig. 6.1. The depth-first search algorithm will start from the root node which is node number 1. It will add this to the stack and immediately pop it as it has been visited. The child nodes of the root node, which are node number 2 and 3, will be added to the stack. The algorithm will first visit node number 2 and pop it from the stack. Now the algorithm will add the child nodes of node number 2, which are nodes 4 and 5 into the stack. The stack will now visit node number 4 and pop it from the stack. Since node number 4 does not have any children, it will visit the next node in the stack which is node number 5 and pop it from the stack. Since node number 5 does not have children, it will visit and pop the next node in the stack which is node number 3. The algorithm will now add the child nodes of node number 3, which are nodes 6 and 7 into the stack. Now, the algorithm will visit node number 6 and pop it from the stack. The algorithm will add the child node of node number 6, which is node 8 into the stack. It will now visit node number 8 and pop it from the stack. Since node number 8 does not have any children, the algorithm will visit and pop the next node in the stack which is node number 7. Now, the child node of node number 7, which is node 9, will be added to the stack. Finally, the algorithm will visit node number 9 and pop it. Since node number 9 does not have any children, the stack will try to visit the next node and pop it. However, the stack is now empty. This means that the entire tree has been traversed and the algorithm will now terminate. From the tracing of the depth-first search algorithm above, the algorithm traversed the nodes in the order (1, 2, 4, 5, 3, 6, 8, 7, 9).

Now, let us assume that a tree is a dungeon. Where each node is a room, and each edge is a path that connects the rooms in the dungeon. If the starting room of the dungeon is considered the root, the rooms surrounding it will be the adjacent vertices and will be added to the stack. From there, one of the rooms is randomly selected to continue the depth-first search and the cycle continues.

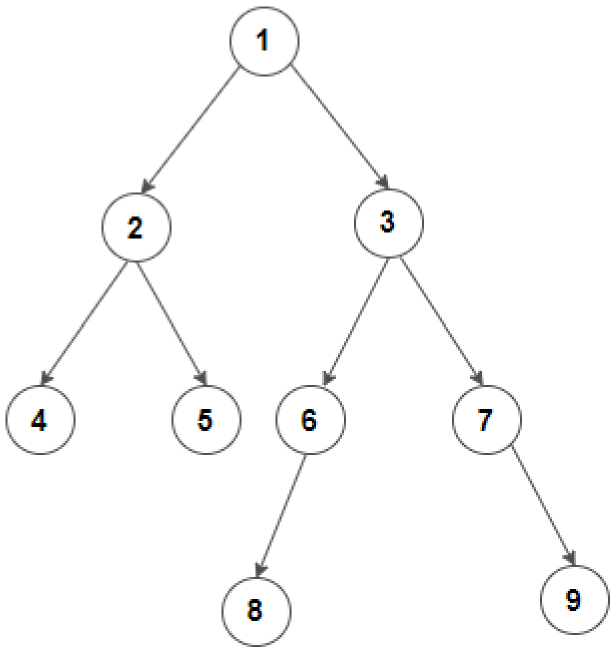


Figure 6.1: Example of a Tree [4]

6.3 Breadth-First Search

Breadth-first search is a graph traversal algorithm that visits all the nodes in a graph by visiting the neighboring nodes at a given depth before moving on to visit the nodes deeper in the graph [8]. While the depth-first search algorithm and the breadth-first search algorithm aim to achieve the same goal, it functions differently and is implemented using a queue. The algorithm starts by enqueuing the starting node and it marks it as visited. While the queue is not empty, the algorithm dequeues a node. After the node is dequeued, the algorithm visits its neighbors and enqueues them. The cycle repeats until the nodes in every depth of the tree have been visited. The time complexity of the breadth-first search algorithm is O (N + E) for the same reasons as the depth-first search algorithm [8].

Now let us apply this algorithm to the same tree in fig. 6.1 that was used in the example for the depth-first algorithm. The breath-first search algorithm starts from the root node, which is node number 1. The root node will be added to the queue. The root node will then be dequeued and marked as visited. After this, the algorithm will add both of its child nodes to the queue, which are nodes number 2 and 3. Now, the next node in the queue is node number 2. This node will be dequeued and marked as visited. Its child nodes, which are nodes number 4 and 5 will now be added to the queue. Now, the next node in the queue is node number 3. This node will be dequeued and marked as visited. Its child nodes, which are nodes number 6 and 7 will be added to the queue. Next, node number 4 will be dequeued and marked as visited. This node does not have child nodes, so no changes will be made to the queue. Now, the next node in the queue is node number 5. This node will be dequeued and marked as visited. Since this node does not have child nodes, no changes will be made to the queue. The next node in the queue is node number 6. This node will be dequeued and marked as visited. Its child node, which is node number 8 will be added to the queue. This node does not have a child node so the algorithm will move on to the next node in the queue which is node 7. This node will be dequeued and marked as visited. Node number 9, which is the child of node number 7, will be added to the queue. Now, the next node in the queue is node number 8. This node will be dequeued and marked as visited. Since this node does not have any children, the algorithm will move on to the next node in the queue, which is node number 9. Node number 9 will be dequeued and marked as visited. Since the queue is now empty, this means that the algorithm has visited every node and will terminate. From the tracing of the breadth-first search algorithm above, the algorithm traversed the nodes in the order (1, 2, 3, 4, 5, 6, 7, 8, 9).

This algorithm will also be used to generate a procedural dungeon generator. By looking at the differences between the two algorithms. We can expect the dungeons that will be generated to have different characteristics in their structure.

6.4 Backtracking Algorithm

Backtracking is an algorithmic technique that is used to find the solutions to a problem by incrementally building a candidate that can lead to a solution for the problem. The algorithm checks whether the candidate can be completed to create a valid solution. If it does not lead to a valid solution, it is rejected, and the algorithm will continue to search for a new candidate that can solve the problem. The algorithm will continue to run until a valid solution has been found or if all the possible candidates do not lead to a solution [16] This process of searching for the solution can be done iteratively or recursively.

One of the key advantages of backtracking algorithms is their ability to find all possible solutions to a problem, as opposed to finding only a single solution. However, this also means that backtracking algorithms can be computationally inefficient. There are various optimization techniques that can be used to improve the efficiency of backtracking algorithms. Some of them include pruning the search space, using heuristics to guide the search, and using dynamic programming [16]

Backtracking is important in the scope of this thesis, as it will be used alongside the depth-first search algorithm to create a procedural dungeon generator. Backtracking algorithms and depth-first search are closely related. In many ways, backtracking is a form of depth-first search that is specifically used to search for solutions to problems. Both algorithms use a stack or recursion to keep track of visited vertices (nodes) or solutions and can search through a large, complex search space. However, backtracking algorithms are specifically designed to solve problems by exploring all possible choices by utilizing depth-first search, while depth-first search is a general algorithmic technique that can be used for a wide variety of problems.

Backtracking algorithms have various applications such as puzzle solving, game playing, optimization, and machine learning. Some examples of problems that can be solved using backtracking algorithms include the Eight-Queens problem, the Sudoku puzzle, the graph coloring problem, and the traveling salesman problem.

Depending on the problem being solved, a backtracking algorithm can make changes in the search space. This is important because the backtracking algorithm that is used in the procedural dungeon generator is non-destructive. This means that the algorithm will not undo or remove choices that have already been made if it is invalid. This is different from the Eight-Queens problem that is covered below. In the Eight-Queens problem, the algorithm is destructive, and it will undo the choices it considers invalid.

6.4.1 Eight-Queens Problem

The Eight-Queens problem is a well-known puzzle in mathematics and computer science that involves placing eight queens on an 8x8 chessboard such that no two queens can attack each other. In other words, we can say that no two queens can be placed on the same row, column, or diagonal of the chessboard [14]. Solving the Eight-Queens problem requires finding all the possible solutions. There are 92 distinct solutions to this problem, which can be obtained by using a backtracking algorithm [2].

The backtracking algorithm is a common approach for solving the 8 Queens Problem. This algorithm works by systematically trying different configurations of the queens on the chessboard and undoing any moves that violate the problem's constraints. The algorithm continues until all possible solutions have been found.

Here is a step-by-step rundown of how the backtracking algorithm solves the problem. First, the algorithm starts with an empty chessboard, Next, the algorithm will place the first queen in the first row and first column of the chessboard. Now, the algorithm will move to the second column and attempt to place the second queen in each row until a valid position that meets the problem's constraint is found.

If no valid position is found in the entire column, the algorithm will backtrack to the previous column and try the next row. The algorithm will continue this process for all the eight queens until a solution has been found. Once a solution has been found, the algorithm can stop and report it, or it can continue to find additional solutions to the problem [6].

CHAPTER 7

Procedural Dungeon Generator

In chapters 3 and 4, we observed that the procedural generation of dungeons follows the constructive approach. This is because dungeons have a rigid structure that can easily be defined. The constructive approach to generating a dungeon has two major steps. The first step revolves around the spatial representation of the dungeon and its other rules such as room types, items, etc. From the generators created by Gellel et al. and Werneck et al., we observed that a grid such as a cellular automaton or a matrix is commonly used to represent the space of a dungeon. The second step revolves around an algorithm or a game engine that will create the layout of the dungeon based on the spatial representation and rules presented in the first step. The generators made by the researchers mentioned previously, both made use of the Unity game engine to create the layout of the dungeon. Using this information, we developed a fully functional procedural content generator. This section will be discussing how the procedural dungeon generator was created, alongside how the dungeon created was used to create a 3D representation.

7.1 Overall Dungeon Characteristics

Before we dive into the procedural dungeon generating algorithm, there are certain rules, constraints, and assumptions that have been taken into consideration in order to generate a dungeon. This section will discuss the content representation of the dungeon by providing a definition of a dungeon and its rooms in the context of the generator, including its important characteristics.

A dungeon in the context of this generator is a two-dimensional grid that consists of a sequence of rooms that are connected to each other. The fig. 7.1 below is an example of a 7 by 7 dungeon. It has a similar core structure to the dungeons seen in older roguelike games, which include rooms and pathways that connect the rooms. Despite these similarities, the graphical representation will be different because the sizes of the rooms and doorways will be assumed to be uniform. Each cell in the grid of the dungeon has the possibility of being a room, and each room has a square shape. This means that a room can have a doorway connecting it to other rooms on any of its four sides. It is important to keep in mind that there is a possibility that a room can fall along the edges of the grid. If this is the case, it cannot be connected to rooms on all four of its sides. For example, if we look at the second grid below in fig. 7.2, we can see that any room situated in the green area has four neighboring rooms around it while the rooms highlighted in yellow only have three neighboring rooms around them, and finally, corner rooms highlighted in red only have two neighboring rooms.

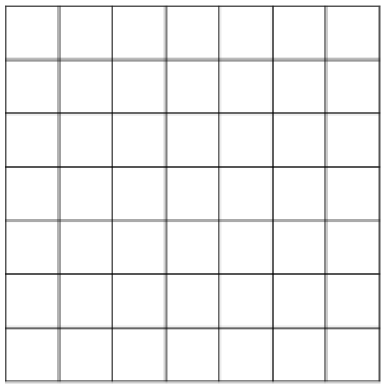


Figure 7.1: Example Grid [26]

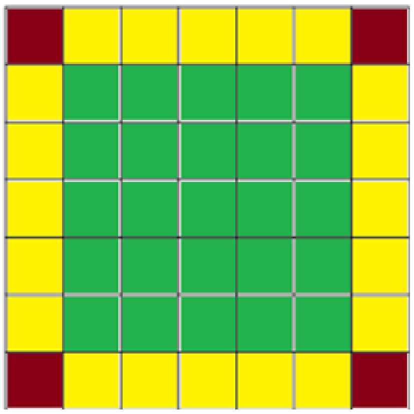


Figure 7.2: Grid Highlighting Keys Areas

From this, we can say that most rooms will be connected to at least two other rooms. The first one is the starting room where the player originally came from, and the second one is another room that is connected through a doorway on any of its other sides. Since a room can be connected to multiple other rooms, it allows for the creation of dungeons with branching pathways. There is also a possibility that a room is only connected to one room. In this case, the room would be a dead-end. The doors of a room are bidirectional, meaning that a player can always go back to the room they came from. This is important as it allows players to backtrack and explore another pathway if they get stuck in a dead-end, without keeping them stuck on a softlock.

The dungeon has one starting room that is randomly selected by the generator. The dungeon also has a single final room which serves as the end goal. The final room is assigned to the last room that is generated. The final room of a dungeon is important because the player needs a reason to explore the dungeon. The final room can also serve as a location to include boss enemies or a connection to another level. The dungeon has 6 types of rooms. Room types are assigned to create an engaging gameplay loop.

The first type is the starting room, which is represented by an S. As the name suggests, it is the room where the player begins their journey. The second type is the treasure room, which is represented by a T. The purpose of this room is to give the player an item that will aid their journey. The third type is the enemy room, which is represented by an E. This room challenges the player and serves as an obstacle in their journey. The fourth type is the empty room, which is represented by an N. This is a filler room that can be used as a place to rest. The fifth type of room is the cursed room, which is represented by a C. This room gives the player a curse or misfortune, which sets them back during their journey. Finally, the sixth type is the final room, which is represented by an F. This room is the end destination that the players are supposed to reach. The room types are assigned in the sequence shown above and repeats. If the previous room was the starting room, the next room is a treasure room, and so on. All rooms except the starting room and final room will be repeated in the cycle as there can only be one starting room and one final room. There is another room type that is an exception in this cycle, which is the cursed room which occurs once every five rooms. Currently, this is hardcoded, but it is possible to make it more interesting by changing its occurrence using an increasing probability. For example, we can make it so that the probability that the next room is a cursed room increases as more rooms are generated. The probability resets once a cursed room is generated and the cycle repeats. The number of rooms to be generated is given to the generator as input from the user.

7.2 Dungeon Generation Algorithm

Now that we understand the definition of a dungeon and its characteristics in the context of this generator, we can now go over our roadmap to create a procedural content generator. First, we need to choose how we are going to spatially represent the dungeon. Next, we need to find an algorithm that can place the rooms in an appropriate sequence. Once the dungeon has been generated, we need to graphically represent the created dungeon. Finally, we need to create a 3D dungeon using the output of this generator using Unity. The dungeon generator was coded using Java in the Eclipse IDE. Java was chosen because it is an object-oriented programming language that I am familiar with. Furthermore, it also resembles the C# programming language which is used in Unity. The Unity game engine was chosen for the 3D representation because it is easier to use compared to other alternatives.

First, we need a spatial representation of the dungeon. We chose a two-dimensional array to represent the grid space of the dungeon in the algorithm because it has a structure that matches the grid shown in fig. 7.1 and defined in the previous section. The size of the array is related to the number of rooms the user wants to generate, and the algorithm used. However, the area of the array will always be greater than or equal to the number of rooms being generated. Each index of the array represents a potential room within the dungeon. The rooms in the dungeon are represented as objects. A room object stores critical information about its properties such as the number of doors, the location of the doors, and the room type. By default, each room in the array is initialized as inactive. This information is stored in the room object using a boolean variable. The dungeon generation algorithm determines which of the rooms to activate and in which order to create the dungeon. Only the rooms that have been activated by the generation algorithm form the dungeon. Suppose that the 7 by 7 grid shown below in fig. 7.3 represents a dungeon array. The deactivated rooms are highlighted in gray and the activated rooms are highlighted in green. Even though every cell is a room object, only the 6 rooms highlighted in green make up the actual dungeon.

Now that we have a spatial representation for the dungeon, we need to find an algorithm that will activate the rooms in an appropriate sequence to generate a dungeon. The goal of the algorithm is simple. It needs to be able to place

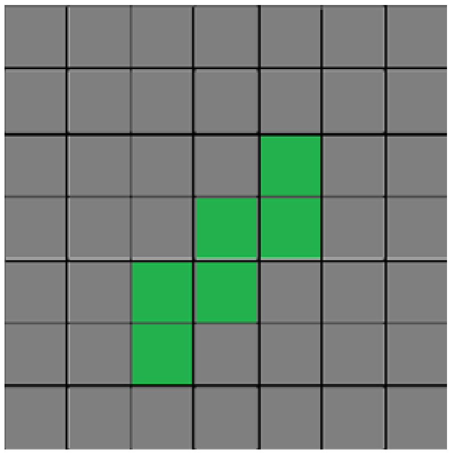


Figure 7.3: Example of a Complete Dungeon Within a Grid

the required number of rooms in a grid, such that there is a complete pathway connecting all the rooms in an accessible manner. A complete pathway can be understood as a continuous path from the starting room S to any room R within the dungeon. In other words, we can say that there needs to be a complete pathway such that a player can visit every room in the dungeon, starting from the starting room. Below are two examples, the dungeon below in fig. 7.4 is continuous and the dungeon in fig. 7.5 is disjoint. In both dungeons, the starting room is room number one. In the continuous dungeon, we can see that there is a clear path connecting all five rooms. However, in the disjoint dungeon, we can clearly see that there is no pathway connecting room number 6 to the rest of the dungeon.

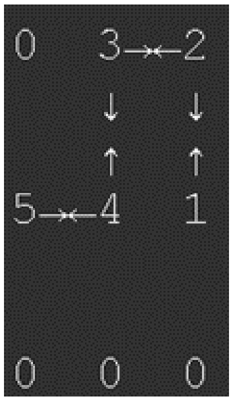
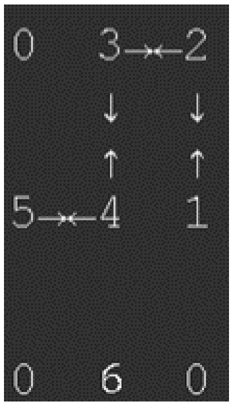


Figure 7.4: Example of a Continuous Dungeon



```
0  3→←2
   ↓  ↓
   ↑  ↑
5→←4  1



0  6  0
```

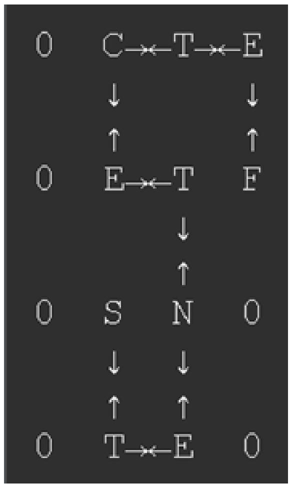
Figure 7.5: Example of a Disjoint Dungeon

To create dungeons like the continuous one seen above, there are two algorithms we can use. These algorithms are depth-first search and breadth-first search. Information about how these two algorithms work within this generator is discussed later in this chapter. These two algorithms achieve the same goal but operate differently and create dungeons that have visually different characteristics. Depending on which algorithm we choose, the information stored by the room object changes.

Finally, we need to create a graphical representation of the dungeon which highlights its traversal and pathways. The printing is divided into two parts. The first part is creating a graphical representation of the two-dimensional dungeon in the form of text. The second part is creating a text file with the appropriate information required by Unity to create a three-dimensional representation of the dungeon. For both parts, we need information about where the rooms are located in the dungeon array, the type of the rooms, and the layout of the rooms. This information for each room is assigned when they are activated by the generator's algorithm.

For the first part of the representation, we can utilize the information provided by the completed dungeon array. Using the dungeon array, we can determine which rooms have been activated, the type of the activated rooms, and the direction of each existing door for each room. The direction of the door determines the layout of each room. Two examples of a fully defined room are: a cursed room with two doors opposite each other, going north and south and a treasure room with two doors adjacent to each other, going north and east.

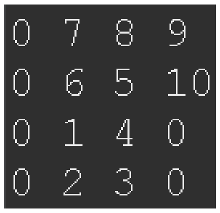
The direction of the door is represented using a Unicode arrow symbol. The arrows help us visualize the pathways of the dungeon. The printing of the dungeon is handled by a function specifically created for this purpose. Below in fig. 7.6 is an example of the visual representation of a dungeon that has 10 rooms in a 4 by 4 grid. Here, the letters represent the type of room, and the arrows represent the pathways of the dungeon. There are pathways pointing forwards and backward to represent bidirectionality. The 0s in the grid are rooms that are inactive. We can see that the algorithm successfully generated a dungeon with complete pathways.



```
0  C→←T→←E
   ↓     ↓
   ↑     ↑
0  E→←T  F
      ↓
      ↑
0  S  N  0
   ↓  ↓
   ↑  ↑
0  T→←E  0
```

Figure 7.6: Example of a 2D Dungeon Representation

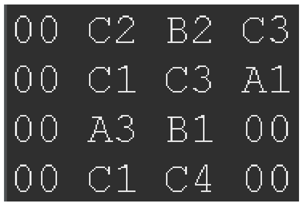
For the second part of the representation, we need to create a text file that can provide the Unity game engine with the appropriate information it requires. We have to produce the same information used to create the two-dimensional graphical representation, but in such a way that the Unity script can easily read it. We cannot simply send the complete dungeon array to the Unity game engine as they are not directly linked. To send the information, we will make use of three other two-dimensional arrays. The first two-dimensional array will store integers. This array provides information about the order in which these rooms were created. The image below in fig. 7.7 showcases how the array will look like for the dungeon in fig. 7.6 that was generated above. The number will be on the index of the respective room. In the example below, 1 will be the starting room and 10 will be the final room.



```
0 7 8 9
0 6 5 10
0 1 4 0
0 2 3 0
```

Figure 7.7: Example of The Room Generation Order Array

The second array stores strings and it provides information about a room's layout. Similar to the last array the information will be on the index of the respective room. The image below in fig. 7.8 showcases how the array will look like for the example dungeon that was generated above in fig. 7.6. More information about how the room layout is represented will be covered in the next section.

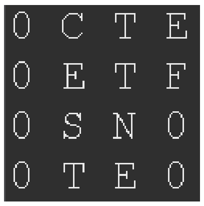


```
00 C2 B2 C3
00 C1 C3 A1
00 A3 B1 00
00 C1 C4 00
```

Figure 7.8: Example of The Room Layout and Orientation Array

Finally, the last array provides information about the type of room. The image below in fig. 7.9 showcases how the array will look like for the example dungeon generated above in fig. 7.6.

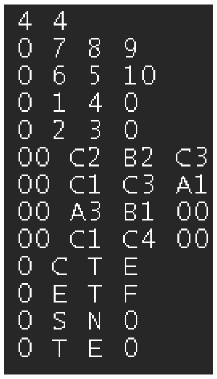
The first line of the text file specifies the number of rows and columns of the grid of the dungeon. After this has been included, the three arrays mentioned above are



```
0 C T E
0 E T F
0 S N 0
0 T E 0
```

Figure 7.9: Example of The Room Type Array

produced and concatenated in the same order they are presented. Below in fig. 7.10 is the contents of the text file for the example dungeon that was shown in fig. 7.6.



```
4 4
0 7 8 9
0 6 5 10
0 1 4 0
0 2 3 0
00 C2 B2 C3
00 C1 C3 A1
00 A3 B1 00
00 C1 C4 00
0 C T E
0 E T F
0 S N 0
0 T E 0
```

Figure 7.10: Example of a Text File

7.3 3D Dungeon Representation

Now that we have created a generator that can make a two-dimensional layout of a roguelike dungeon, we can use the information obtained from it to create a 3D representation in Unity. There are three important steps to creating a 3D representation. First, the required information needs to be extracted from the text file. Next, there need to be prefabs that represent each of the different layouts of rooms. Finally, using the information gained from the text file, we can create the 3D representation by initializing the correct room prefab to the correct location with the correct color. This section discusses these steps in detail.

The text file produced by the dungeon generator will be given to the Unity program as input. The content of the text file is read by the Unity script and stored in an array. Each index of this text array represents a line in the text file. The column and row size of the dungeon will be extracted from the first line of the text file and stored in integer variables. The row and column size will be used to create three two-dimensional arrays that will store information from the input in the text file. The first array stores the location of each room, the second array stores the layout of each room, and the third array stores the type of each room. After the first line, each line represents a row of the text array. Since the script knows the size of the dungeon, it will loop through each array by going over the correct number of lines. Since each index of the text array is a line, the line needs to be split based on blank spaces. Once the line is split, we can access each individual element of the three arrays. Each element is stored in the corresponding index of the array. This process will be repeated for all three arrays in the text file until all the information has been extracted.

Before the rooms can be generated, we need to instantiate prefabs. Prefabs are 3D models that represent objects within a video game. The room prefabs are square and were created for this project using the existing tools within Unity. One of our assumptions about a room is that it will be connected to at least one other room and as many as four other rooms. This means that there are 15 possible room layouts that need to be represented by the room prefabs. The Unity program only has 5 prefabs representing a subset of every possible room layout. Rooms with a single door are called type A rooms and the door can be on any of the four sides. Rooms with two doors on opposite sides of each other are called type B rooms. Rooms with two doors on adjacent sides are called type C rooms. Rooms with three doors are called type D rooms and finally, rooms with doors on all sides are called type E rooms. There can only be one configuration for type E rooms. The 5 prefabs are shown below.

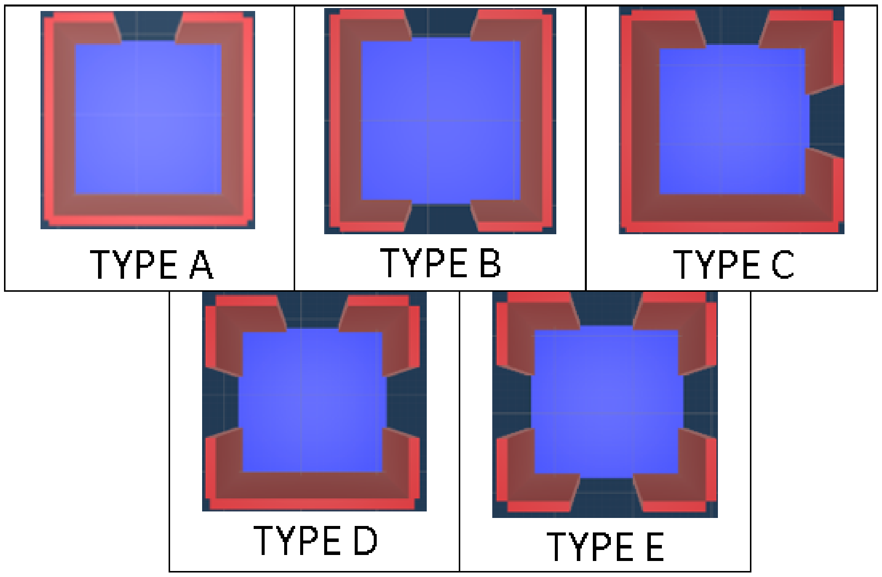


Figure 7.11: The 5 Core Room Layouts

We only need 5 prefabs because the other remaining room layouts are simply variations of these 5 layouts that can be created by rotating the rooms 90 degrees at a time. The fig. 7.12 below displays all the possible room layouts.

Now that we have prefabs, the 3D representation of the dungeon can finally be generated. Using the information stored in the arrays, we can select the correct prefab to place in the correct position with the correct orientation and color. The 3D world space of the Unity game engine uses a three-dimensional cartesian coordinate system with x, y, and z coordinates. To instantiate the room in the correct position, we manipulate only the x and z coordinates. The x-coordinate position is chosen by multiplying the row index of the room by an offset, and the z-coordinate position is chosen by multiplying the column index of the room by an offset. An offset is a number that is used to space out the rooms to prevent them from overlapping. The offset is the length of the room, which is around 20 units. For example, if there is a room at the index [3, 2] we will instantiate an object at the coordinate:

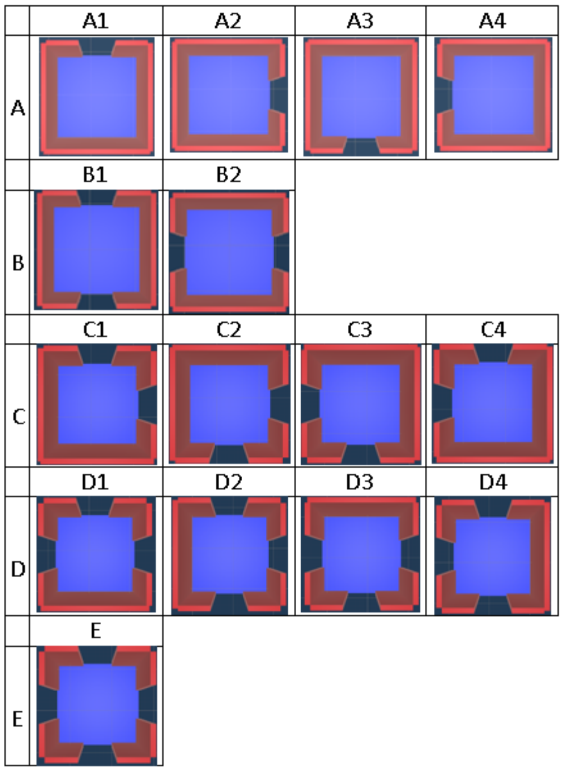


Figure 7.12: All Possible Room Layouts

$$(row * offset,\ 0,\ column * offset)$$

$$(3 * 20,\ 0,\ 2 * 20)$$

$$(60,\ 0,\ 40)$$

Depending on the room's layout, the room prefab may be rotated when instantiated. The rotation is achieved using Quaternion rotations about the y-axis. For example, to create room layout A1, we do not have to rotate, but for room layout A2, we rotate the A prefab by 90 degrees about the y-axis. Similarly, to create the A3 room layout we rotate the A prefab by 180 degrees, and finally for the room layout A4, the room A prefab is rotated 270 degrees. This can be observed in fig. 7.12.

We can use the third array to distinguish the different types of rooms in the 3D representation. They will be distinguished using different colors. The starting room is represented with the color white, the final room with the color black, the treasure rooms with the color green, the enemy rooms with the color yellow, the empty rooms with the color blue and finally the cursed rooms with the color red. Below in fig. 7.13 is an example of a C1 layout starting room.

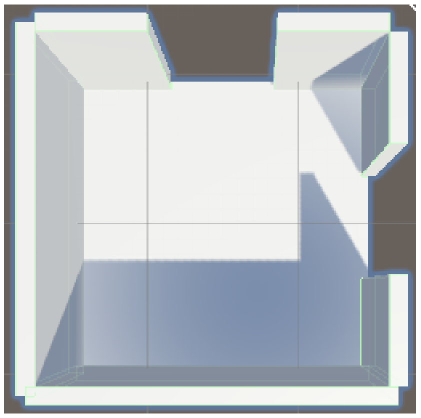
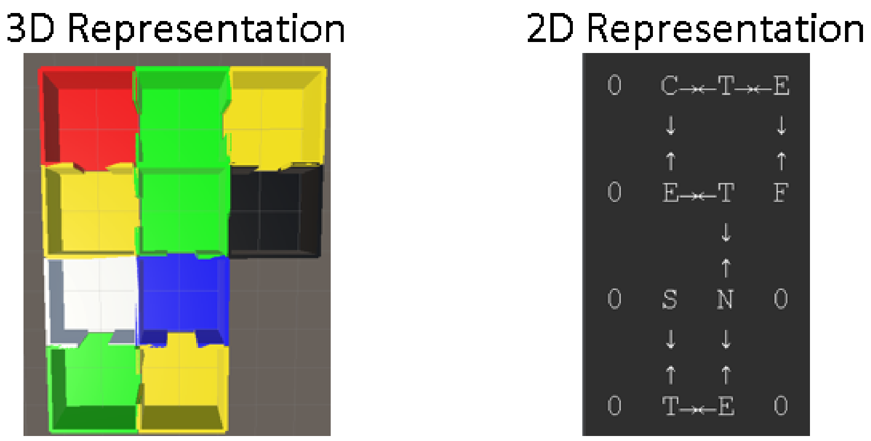


Figure 7.13: Example of a C1 Layout Starting Room

Below in fig. 7.14 is the 3D representation of the dungeon that was shown in fig. 7.6 of the previous section. The 0s in the 2D representation are inactive rooms, this means that the entire left column of the 2D representation has been ignored in the 3D representation.



```
0  C→←T→←E
   ↓     ↓
   ↑     ↑
0  E→←T  F
      ↓
      ↑
0  S  N  0
   ↓  ↓
   ↑  ↑
0  T→←E  0
```

Figure 7.14: Side by Side Comparison of 2D and 3D Representation

7.4 Room Generation Algorithm

Two different versions of the dungeon generator were created. One of the generators was created using the depth-first search algorithm and the other was created using the breadth-first search algorithm. These two algorithms work because we have defined our dungeon such that it is similar to a graph. Each room in the dungeon is a node and the doorways that connect the rooms are the edges. This section goes over the details of how the concept of these two algorithms was used to generate dungeons.

7.4.1 Depth-First Dungeon Algorithm

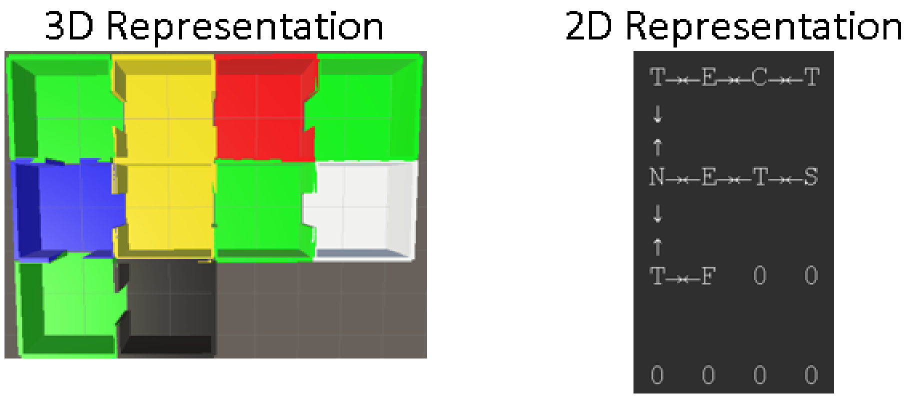
First, the starting room is randomly selected from anywhere within the dungeon array and activated. The size of the dungeon array is calculated by taking the square root of the number of rooms to be generated and adding 1 to it.

$$row = column = \sqrt{No.\ of\ Rooms} + 1$$

This allows the generated dungeon to have a tight space to fit all of the rooms in. The depth-first algorithm can work on any array size as long as the area is bigger than the number of rooms to be generated. Despite this fact, restricting the size of the dungeon array allows for the creation of dungeons with more branching paths. This will be discussed in detail when evaluating the generators in chapter 8.

Now that the generator has a starting point, it will start a loop that will not terminate until the desired number of rooms has been generated. At the start of each loop, the generator will choose a random direction to create the next room. The direction can be up, down, left, or right from the starting room. Once the direction has been chosen, the generator will update its current position and check if the room in this position has already been activated or not. If it has not been activated, the room will be activated and this new room will become the current room. When the room is activated, other information about the room, such as the number of doors it has, its type, and the directions of the doors, will be updated. In this example, the type of the new room will be a treasure room and it will have one door, which points toward the previous room. The information for the previous room will also be updated. In this example, the previous room will now have a door that faces the new room that was created. There are also other edge cases the generator will need to check for. For example, if the current room is at the top left corner of the grid, we cannot create a room above it or to its left. This would cause the generator to go out of the bounds of the dungeon array. When a room in the chosen direction is already active (or does not exist), the generator will simply go back to the previous position and loop until it chooses a direction where an inactivated room is available. Now that the current position has shifted to the newly generated room, this process simply repeats until the required number of rooms has been generated.

While the depth-first dungeon generator is working, there is one fatal flaw. The depth-first search algorithm randomly chooses a direction without thinking about the consequences that could occur in the future. Furthermore, it does not have knowledge of the scope that is outside of the current room and its surroundings. In other words, we can say that it does what it wants and deals with the consequences later. Due to the nature of the depth-first search, there is a possibility that the generator can become stuck in a dead-end, where it cannot progress regardless of which direction it chooses. For example, the generator will become stuck if all the neighboring rooms surrounding the current room are activated. To solve this issue, the generator will need to be able to recognize when it becomes stuck and backtrack to a position where it will not become stuck. To backtrack, the generator will make use of a stack that will keep track of the previous direction when a new room is generated. At the start of each loop, the generator will call a function that will check if it is stuck. The function achieves this by checking if all the rooms surrounding the current room are activated. If all of the surrounding rooms are activated, the function will raise a boolean flag to true and the generator will enter the backtracking state. In this state, the stack containing all previous directions will be popped. Once the stack has been popped, the popped direction is used to go back to a previous room in the dungeon. The generator will backtrack until it reaches a previous room that does have inactivated rooms in its surroundings. After this, the backtracking will stop, and the generator will resume its previous function. Below in fig. 7.15 is an example of a 10-room dungeon created using this algorithm. Again, the 0s in the 2D representation are inactive rooms, and will be ignored in the 3D representation



```
T→←E→←C→←T
↓
↑
N→←E→←T→←S
↓
↑
T→←F   0   0


0   0   0   0
```

Figure 7.15: 10-Room Dungeon Created Using the Depth-First Algorithm

It is interesting to note that every time the generator backtracks, a new branching pathway is created in the dungeon. This can be observed in the example below, as backtracking occurs when the generator gets stuck at the treasure room in the top right corner of the grid. It backtracks to the empty room represented by N and creates a new branching path below it. If we look at the dungeon in terms of a tree, this can be seen as the depth-first search algorithm moving on to traverse another branch in the tree once it has reached the end of a branch.

With the help of the backtracking algorithm, the generator is able to create a dungeon that is 100% playable. This generator is similar to the "Subsection Search on Halt" generator created by Gellel and Sweetser because the generator has two states. In the first state, the generator will continue to generate rooms until it gets stuck. If the generator gets stuck, it will enter the second state where it tries to resolve the issue by backtracking. Once the issue has been fixed, the generator will return to its first state and resume its function. Now, we will be moving on to the breadth-first dungeon algorithm.

7.5 Breadth-First Dungeon Algorithm

Similar to the depth-first algorithm, the starting room is randomly selected from the dungeon array and activated. The size of the dungeon array is calculated by multiplying the number of rooms to be generated by 2.5 and taking its square root.

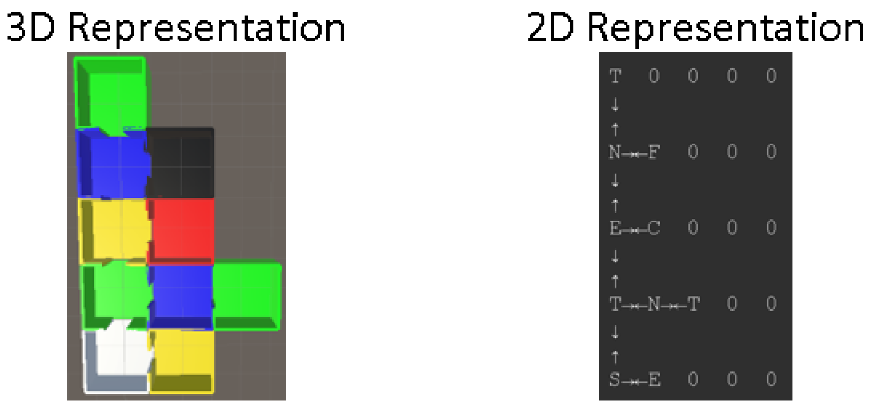
$$row = column = (int)\sqrt{No.\ of\ Rooms\ *\ 2.5}$$

Only the integer value is taken from the result of the calculation. The breadth-first algorithm requires more space and freedom to perform well and without errors. More on this will be covered when evaluating the generators in chapter 8.

Unlike the depth-first algorithm which simply creates the layout of the dungeon, the breadth-first algorithm traverses through a rough sketch of a dungeon layout that has already been created. The rough sketch of the dungeon layout is created by randomly assigning the number of doors and the direction of the doors for each room. Each room will be connected to at least two rooms, and there are checks in place to ensure that rooms with impossible layouts cannot be created. For example, a room in the top right corner of the dungeon array cannot have rooms above it or to its right. This also means that this room cannot have doors to more than two other rooms. The rough layout of the dungeon has two major flaws. The first flaw is the fact that the doors are not bidirectional, and the second flaw is that the dungeon is disjointed. This occurs because the neighboring rooms, alongside the pathing of the dungeon, are not taken into account when randomly assigning the number of doors and the location of the doors. The goal of the breadth-first search algorithm is to traverse through this rough sketch from a randomized starting point and create a complete dungeon by fixing the flaws mentioned above.

While the layout has been created, all the rooms in the dungeon array remain inactive. This is because the generator will search through the layout and activate the appropriate rooms that will create the dungeon. As mentioned before, only the activated rooms are part of the final dungeon. Similar to the depth-first algorithm, the starting room is randomly selected and activated. The starting room will be stored in a queue that stores room objects. Since this generator follows the concepts of breadth-first search, a queue will be used to traverse the rough sketch of the dungeon layout. Now that the layout has been created, the generator will enter a loop that will not terminate until the desired number of rooms has been activated. At the start of the loop, the algorithm will dequeue a room from the queue. This room object is stored inside a temporary variable, and it represents the current position of the generator. We will call this temporary room, the parent room. In the first iteration of the loop, it will be the starting room. The generator now checks if there are doors on each side of the room. If a door to another room exists, it will check if the room it is connected to is activated or not. If it is not activated, the room will be activated, and it will be added to the queue. Rooms activated by the parent room will be called a child room or its children. Unlike the depth-first algorithm, other information about the room is not updated when activated. Furthermore, only the parent room will have the direction of its doors updated to point toward its children. Once all the possible rooms connected to the parent have been activated, the next iteration of the loop will begin. This time the first child room that was added to the queue will be dequeued. This child room will now represent the current position of the generator and will be considered the parent room for the other rooms that will be generated from it. After this, the generator will function in the same way as it did in the first iteration, and it will repeat until the desired number of rooms has been generated.

After the rooms have been generated, the information for each room is finally updated. Each room is assigned a type based on the room layout of their parents. The direction of the doors is also updated such that the child rooms will now have doors that point towards their parent room. Below in fig. 7.16 is an example of a 10-room dungeon created by this algorithm.



```
T   0   0   0   0
↓
↑
N→←F   0   0   0
↓
↑
E→←C   0   0   0
↓
↑
T→←N→←T   0   0
↓
↑
S→←E   0   0   0
```

Figure 7.16: 10-Room Dungeon Created Using the Breadth-First Algorithm

Unlike the depth-first algorithm, the breadth-first algorithm does not have backtracking. This is because the breadth-first algorithm is not limited to knowledge of only its current position. However, to have more information and awareness, the room object has been changed to store more information compared to the depth-first algorithm. For example, the breadth-first algorithm requires its rooms to be aware of their position within the dungeon array. This is because the breadth-first algorithm will need to make sure that all the rooms in the same depth for every branching path have been activated. This causes the current room pointer to constantly move from one branch to another. In this regard, the depth-first algorithm is a lot simpler. Since it traverses one branch of the dungeon at a time, it does not require the rooms to be aware of its position in the dungeon array. The breadth-first dungeon algorithm is similar to the "Persistent Subsection Search" generator created by Gellel and Sweetser as it is constantly running in the same state and does not require backtracking.

CHAPTER 8

Evaluating the Procedural Dungeon Generator

Now that we have two generators that are able to create roguelike dungeons, we need to evaluate their performance and the dungeons they produce. First, we evaluate the performance of the generators based on their different properties such as speed, reliability, and controllability. Next, we will attempt to evaluate the enjoyment factor of the generated dungeons objectively, by using a modified version of the path difference heuristic made by Gellel and Sweetser alongside a new metric called the choice heuristic. Finally, based on the results of our evaluation, we will compare the dungeons generated by both generators and draw some conclusions.

The first thing we can say about the generators is the fact that they are reliable. They are able to generate a dungeon that is 100% complete without running into errors. There is always a connected and continuous path to every room from the starting room. Second, we can say that the generators are fast, as they can generate a dungeon with as many as 10,000 rooms within a short period of time. The generators have some amount of controllability because we can specify the number of rooms to generate. It is possible to create the smallest dungeon with two rooms or a large dungeon with 10,000 rooms or potentially more.

While it is easier to point out the observable differences between the dungeons created by these two generators, the same cannot be said when choosing which of these generators create a better, more enjoyable dungeon. If the generators were part of a fully-fledged game, we could ask a random sample of people their opinion on which of the dungeons they enjoyed exploring more than others. However, this would still be a subjective method of evaluating the dungeon. To attempt to evaluate the enjoyment factor of the dungeon objectively, a modified version of the path difference heuristic made by Gellel and Sweetser will be used alongside a new metric called the choice heuristic.

The path difference heuristic created by Gellel and Sweetser has been modified as the dungeons generated by our generators do not have locks and keys that prevent the player's progression. The calculation of the critical path was modified as it takes the keys and locks into consideration. It has been modified such that it will represent the longest possible route a player can take to reach the final room. To make the calculation of the critical path simpler, it will be calculated by counting every room the player has visited in the dungeon before reaching the final room. The calculation will not include the rooms that the player has backtracked through after reaching a dead-end. This would mean that the critical path will simply be the total number of rooms. The shortest path or the spine will remain the same, and it represents the shortest possible route that can be taken to reach the final room. The calculation of the spine and the critical path will include both the starting and final room of the dungeon. Each room in the route will be considered as one unit in the calculation. The lowest possible value for the path difference heuristic is 0, which occurs when the critical path and the spine are the same. The larger the path difference heuristic, the more explorable and enjoyable a dungeon is. For example, to calculate the modified path difference heuristic for the dungeon in fig. 7.16, we need to calculate the critical path first. The critical path is simply the number of rooms in the dungeon which is 10. The spine will be 5 as the player must go through at least 5 rooms, including the starting and final rooms. Therefore, the modified path difference heuristic for the dungeon is $10 - 5$ which is 5.

The choice heuristic is a metric that will be used to represent the maximum number of choices a player will have to make while exploring a dungeon. The metric is calculated by taking the sum of all the choices a player will need to make when they are forced to choose between multiple doors. For the calculation, we assume that the player will only move forward and will not consider going back to the previous room unless they reach a dead-end. However, the backtracking from a dead-end will not be included in the calculation. Furthermore, if there is a room that presents a player with only a single choice, it will not be included in the calculation. For example, if we want to calculate the choice heuristic for the dungeon in fig. 7.16, we will start looking at the choices presented to the player starting from the starting room. In the starting room S, the player needs to choose between 2 doors, in the treasure room above the starting room, the player needs to choose between 2 doors, in the enemy room above the treasure room, the player needs to choose between 2 doors and finally, in the empty room above the enemy room, the player needs to choose between 2 doors. If we add all the choices, we get a choice heuristic of 8. The rooms without branching paths that present the player with only a single choice were not considered in the calculation.

The modified path difference heuristic and the choice heuristic will be calculated for dungeons of small (2 rooms and 5 rooms), medium (15 rooms and 35 Rooms), and large (80 rooms) sizes for both generators. For the depth-first generator specifically, the medium rooms will also be generated in a larger grid size to demonstrate how the openness or tightness of the grid size leads to the generation of interesting or uninteresting dungeons.

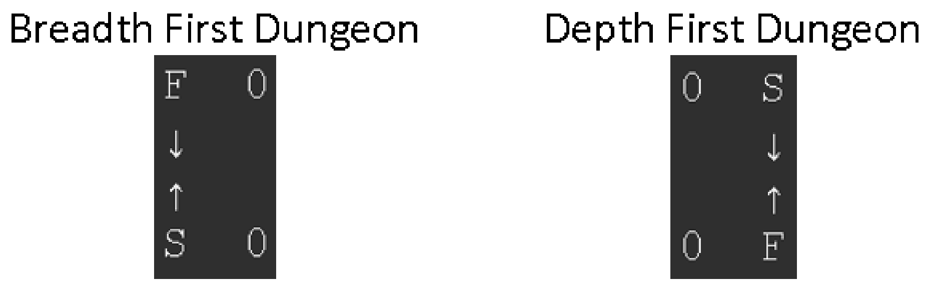


Figure 8.1: 2-Room Dungeons

8.1 Case 1: Small Dungeons

In fig. 8.1 above, the dungeon to the left is a 2-room dungeon generated using the breadth-first algorithm and the dungeon to the right is a 2-room dungeon generated using the depth-first algorithm. For both dungeons, the path difference heuristic is 0 and the choice heuristic is 0. For both generators, the first room will always be the starting room and the second room will always be the final room. The player only has one choice, and it is to move forward into the final room. There is no observable difference that can be seen either. A dungeon with 5 rooms was created as we cannot gain any valuable insight from this.

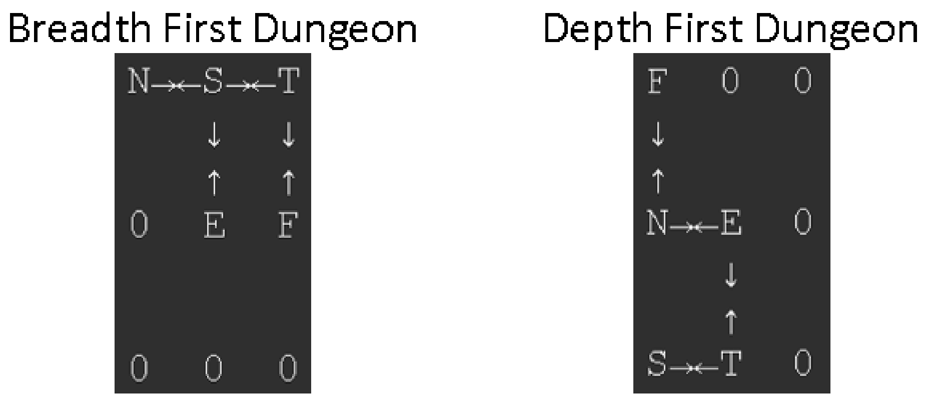


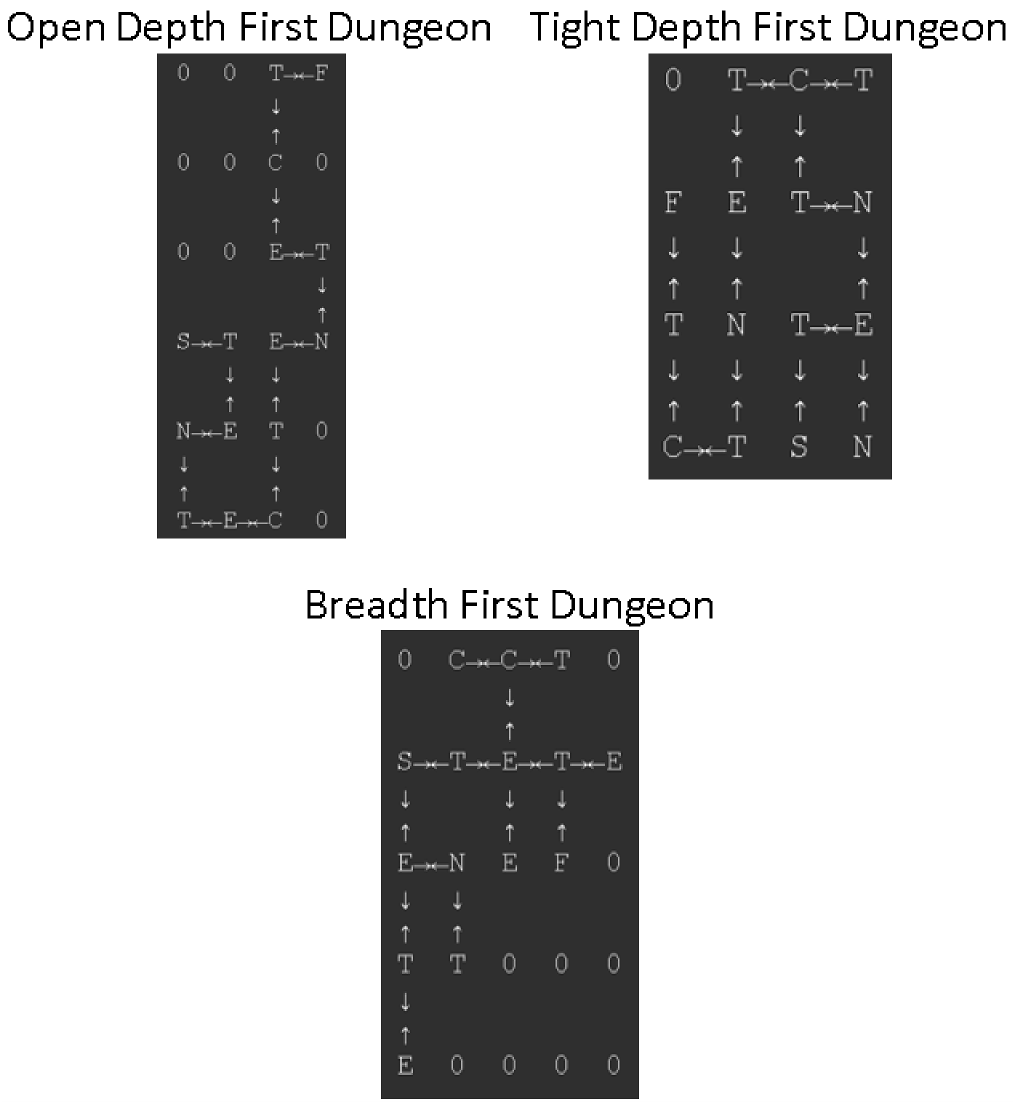
Figure 8.2: 5-Room Dungeons

The dungeon to the left is a 5-room dungeon generated by the breadth-first algorithm and the dungeon to the right is a 5-room dungeon generated by the depth-first algorithm. The dungeon generated by the breadth-first algorithm has a path difference heuristic of 2 and a choice heuristic of 3. On the other hand, the dungeon generated by the depth-first algorithm has a path difference heuristic of 0 and a choice heuristic of 0. The breadth-first dungeon has 3 different branching paths the player can choose from in the starting room. Furthermore, the player is not required to visit every room to reach the final room. While on the depth-first dungeon, the player only has the choice to move forward and is required to visit every room. Even in a small dungeon with 5 rooms, we can see that the breadth-first dungeon provides the player with choices while the depth-first dungeon does not.

8.2 Case 2: Medium Sized Dungeons

In fig. 8.3 below, the dungeon on the bottom is a 15-room dungeon generated by the breadth-first generator, the dungeon on the left is a 15-room dungeon generated by the depth-first generator on a large 9 by 9 grid, and the dungeon on the right is a 15-room dungeon generated by the depth-first generator but on a smaller 4 by 4 grid. The breadth-first dungeon has a path difference heuristic of 10 and a choice heuristic of 11. The 9 by 9 depth-first dungeon has a path difference heuristic of 0 and a choice heuristic of 0. The 4 by 4 depth-first dungeon has a path difference heuristic of 2 and a choice heuristic of 4. We can clearly observe that the breadth-first dungeon presents the player with many choices compared to both depth-first dungeons. If we look at only the depth-first dungeons, we can observe that the dungeon with a tight 4 by 4 grid has more branching pathways compared to the dungeon with an open 9 by 9 grid. While the dungeon with the 9 by 9 grid does have 15 rooms, the explorability and enjoyability of the dungeon are no different than the 5-room depth-first dungeon seen earlier.

In fig. 8.4 below, the dungeon on the bottom is a 35-room dungeon generated by the breadth-first generator, the dungeon on the left is a 35-room dungeon generated



Open Depth First Dungeon

```
0  0  T→←F
      ↓
      ↑
0  0  C  0
      ↓
      ↑
0  0  E→←T
         ↓
         ↑
S→←T  E→←N
   ↓  ↓
   ↑  ↑
N→←E  T  0
↓     ↓
↑     ↑
T→←E→←C  0
```

Tight Depth First Dungeon

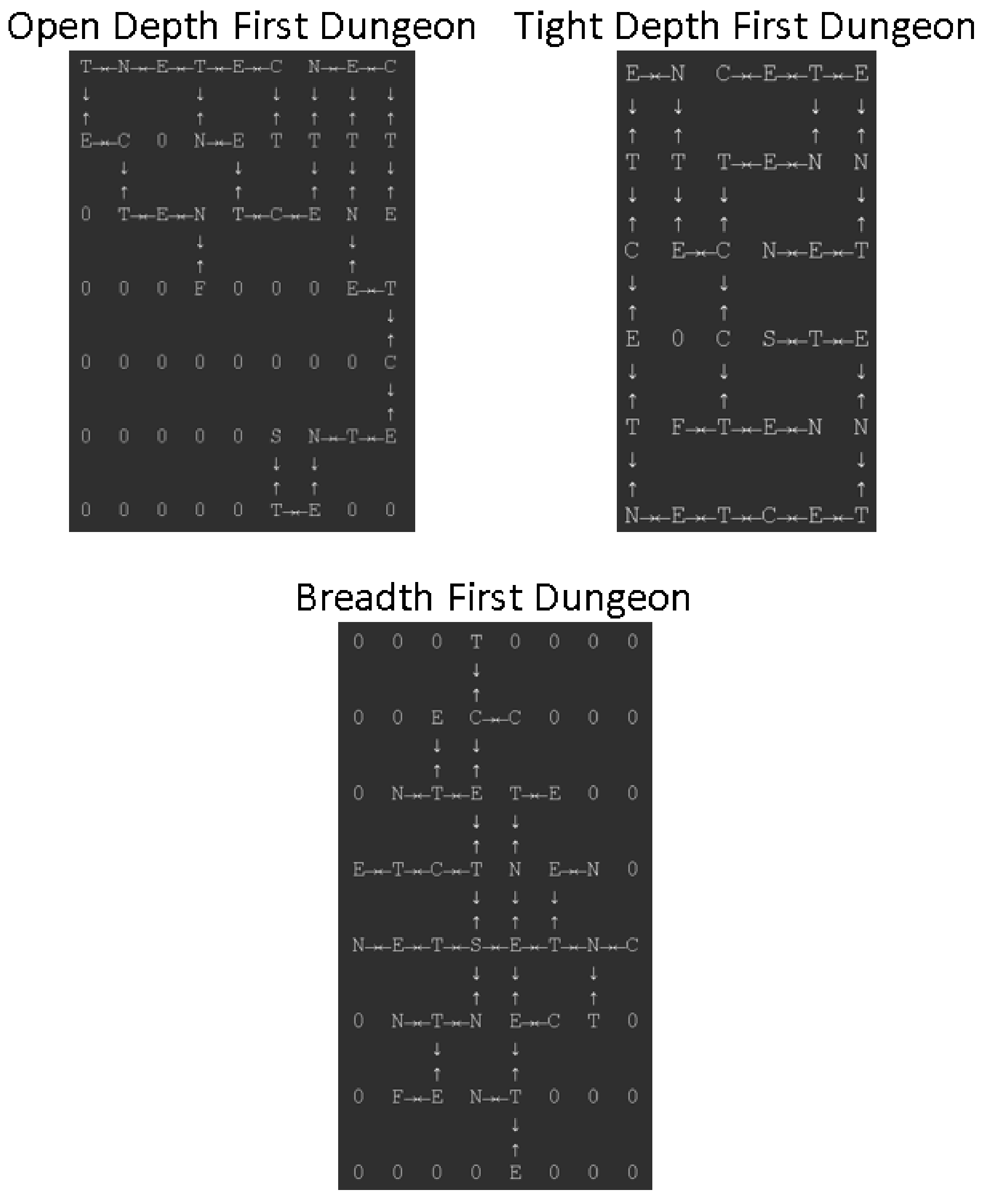
```
0  T→←C→←T
   ↓  ↓
   ↑  ↑
F  E  T→←N
↓  ↓     ↓
↑  ↑     ↑
T  N  T→←E
↓  ↓  ↓  ↓
↑  ↑  ↑  ↑
C→←T  S  N
```

Breadth First Dungeon

```
0  C→←C→←T  0
      ↓
      ↑
S→←T→←E→←T→←E
↓     ↓  ↓
↑     ↑  ↑
E→←N  E  F  0
↓  ↓
↑  ↑
T  T  0  0  0
↓
↑
E  0  0  0  0
```

Figure 8.3: 15-Room Dungeons

by the depth-first generator on a 9 by 9 grid, and the dungeon on the right is a 35-room dungeon generated by the depth-first generator on a 6 by 6 grid. The breadth-first dungeon has a path difference heuristic of 30 and a choice heuristic of 24. The depth-first dungeon with a 9 by 9 grid has a path difference heuristic of 6 and a choice heuristic of 4. The depth-first dungeon with a 6 by 6 grid has a path difference heuristic of 13 and a choice heuristic of 6. The breath-first dungeon continues to have more choices and branching pathways compared to the depth-first dungeons. If we look at only the depth-first dungeons, the dungeon with the smaller



```
T →← N →← E →← T →← E →← C    N →← E →← C
↓              ↓         ↓    ↓    ↓    ↓
↑              ↑         ↑    ↑    ↑    ↑
E →← C    0    N →← E    T    T    T    T
     ↓              ↓         ↓    ↓    ↓
     ↑              ↑         ↑    ↑    ↑
0    T →← E →← N    T →← C →← E    N    E
               ↓                   ↓
               ↑                   ↑
0    0    0    F    0    0    0    E →← T
                                        ↓
                                        ↑
0    0    0    0    0    0    0    0    C
                                        ↓
                                        ↑
0    0    0    0    0    S    N →← T →← E
                         ↓    ↓
                         ↑    ↑
0    0    0    0    0    T →← E    0    0
```

```
E →← N    C →← E →← T →← E
↓    ↓              ↓    ↓
↑    ↑              ↑    ↑
T    T    T →← E →← N    N
↓    ↓    ↓              ↓
↑    ↑    ↑              ↑
C    E →← C    N →← E →← T
↓         ↓
↑         ↑
E    0    C    S →← T →← E
↓         ↓              ↓
↑         ↑              ↑
T    F →← T →← E →← N    N
↓                        ↓
↑                        ↑
N →← E →← T →← C →← E →← T
```

```
0    0    0    T    0    0    0    0
               ↓
               ↑
0    0    E    C →← C    0    0    0
          ↓    ↓
          ↑    ↑
0    N →← T →← E    T →← E    0    0
               ↓    ↓
               ↑    ↑
E →← T →← C →← T    N    E →← N    0
               ↓    ↓    ↓
               ↑    ↑    ↑
N →← E →← T →← S →← E →← T →← N →← C
               ↓    ↓         ↓
               ↑    ↑         ↑
0    N →← T →← N    E →← C    T    0
          ↓         ↓
          ↑         ↑
0    F →← E    N →← T    0    0    0
                    ↓
                    ↑
0    0    0    0    E    0    0    0
```

Figure 8.4: 35-Room Dungeons

grid size continues to have more choices compared to the dungeon with the larger grid size.

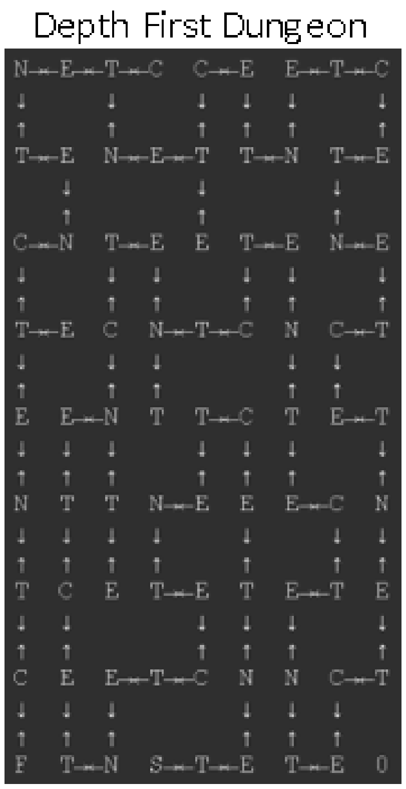
For both generators, we have observed that an increase in the number of rooms increases both the path difference heuristic and the choice heuristic. However, the breadth-first dungeons continue to score significantly higher on both metrics. We have also observed that a depth-first dungeon with a tight grid scores higher on both metrics compared to a depth-first dungeon with an open grid.

8.3 Case 3: Large Dungeons

In fig. 8.5 below, the dungeon below is an 80-room dungeon generated by the breadth-first generator and the dungeon above is an 80-room dungeon generated by the depth-first generator. The breadth-first dungeon has a path difference heuristic of 71 and a choice heuristic of 58. The depth-first dungeon has a path difference heuristic of 6 and a choice heuristic of 10. Again, the breadth-first dungeon has a significantly higher score on both metrics. The path-difference heuristic of the depth-first dungeon was surprisingly low even though it was created in a tight grid. Furthermore, the path-difference heuristic is lower than the 35-room dungeon in fig. 8.4, but the choice heuristic is still higher compared to it.

8.4 Evaluation Discussion

From the analysis above, the first notable information obtained was the differences in the number of dead-ends and branching pathways the dungeons generated by the depth-first and breadth-first generators have. When the dungeon has 2 rooms the differences between the dungeons are not observable. However, even in a small dungeon with 5-rooms, the differences in the structure of the dungeon and the metrics can be observed. Furthermore, the differences only get larger and more visible as the number of rooms in the dungeon increases. The differences in the number of branching pathways are captured by the choice heuristic. For example, the 35-room dungeon generated by the breadth-first algorithm in figure fig. 8.4 has a choice heuristic of 24, while the 35-room dungeon generated in a tight grid size



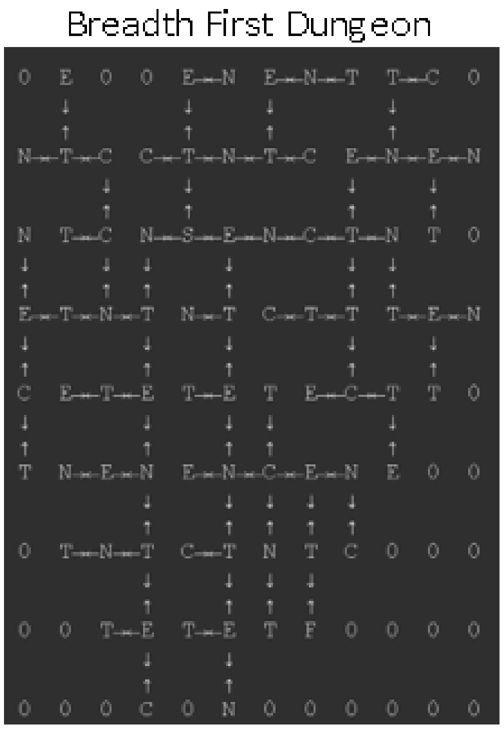


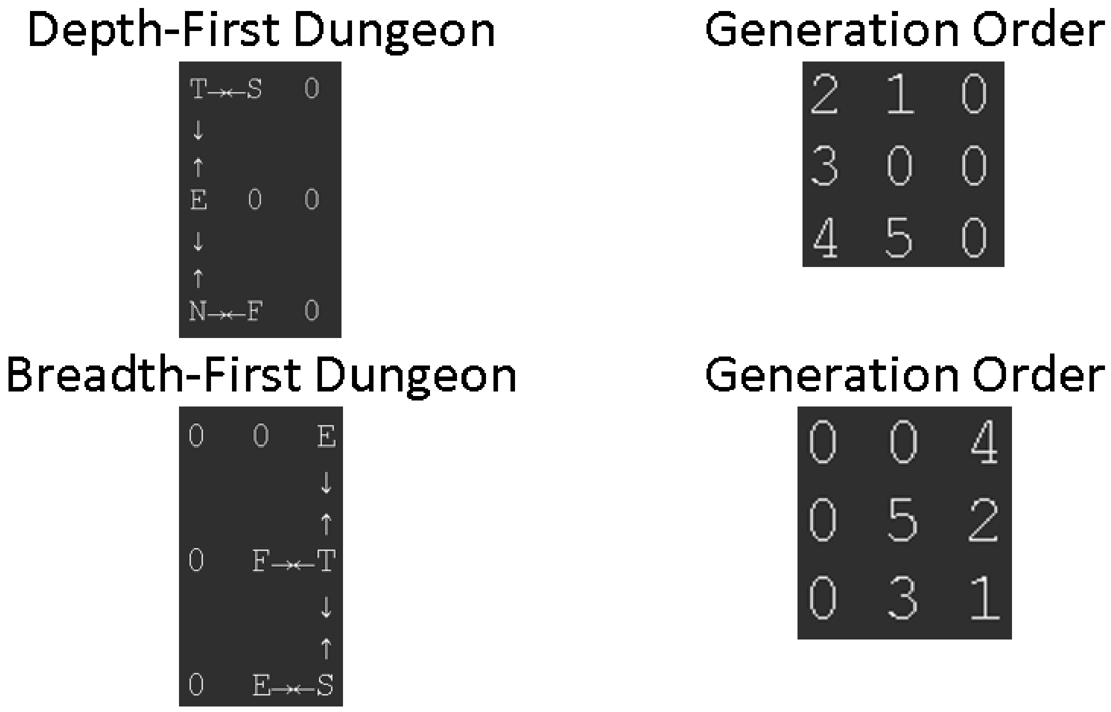
Figure 8.5: 80-Room Dungeons

by the depth-first algorithm in fig. 8.4 has a choice heuristic of 6. This difference can be attributed to the fact that the depth-first generator only creates a branching pathway when backtracking occurs, while in the breadth-first generator, each room has the possibility to create a branching pathway.

The second notable information is the changes in the characteristics and the metrics of the depth-first dungeons that occur due to the size of the dungeon array. If the depth-first generator is given a large array to generate a small number of rooms, there is a high probability that the generated dungeon will not backtrack even once. The resultant dungeon will have long linear pathways that lead directly to the final room without the existence of any branching paths. The player would not have the opportunity to choose and explore different paths. This is also reflected in the modified path difference heuristic and choice heuristic scores. For both the 15-room and the 35-room dungeons, the dungeon with the smaller grid size scored higher on both metrics. To make the depth-first dungeons as enjoyable as possible, the size of the dungeon array is adjusted to be as tight as possible based on the number of rooms required to be generated.

On the other hand, the breadth-first dungeon does not perform well with a restricted dungeon array size. A smaller grid size causes the breadth-first algorithm to run into an error occasionally. This error is caused by the incomplete nature of the rough dungeon layout. When the grid is small, many rooms will fall on the edges of the grid. The edges are the red and yellow areas of the grid as seen in fig. 7.2 in chapter 7. The edges can only be connected to a limited number of rooms, and this can cause the rough dungeon layout to become disjointed. While the breadth-first algorithm is going over this rough layout, the queue will be empty at some point even though it has not activated the required number of rooms. An error occurs due to the generator attempting to dequeue an empty queue. This error is more commonly seen when creating a dungeon with a smaller number of rooms. This error is also the reason why we were not able to provide examples of breadth-first dungeons created using a smaller grid size.

The third noticeable information is the difference in the order of room types that can be observed between the breadth-first and depth-first dungeons. Due to the nature of the depth-first search algorithm, the order of the room types is predictable and it clearly follows the pattern that was outlined at the beginning of chapter 7. If we look at the depth-first dungeon and its room generation order in fig. 8.6, we can see that the connection of the rooms is sequential. This means that room 2 will always be connected to room 1 and room 3 will always be connected to room 2, and so on. Since the connection of the rooms is sequential, the order of the room type is also sequential and will follow the outlined pattern.



```
T→←S  0
↓
↑
E  0  0
↓
↑
N→←F  0
```

```
2 1 0
3 0 0
4 5 0
```

```
0  0  E
      ↓
      ↑
0  F→←T
      ↓
      ↑
0  E→←S
```

```
0 0 4
0 5 2
0 3 1
```

Figure 8.6: Dungeons and Their Room Generation Order

While the breadth-first dungeons also follow the correct order, the nature of the breadth-first search algorithm causes the pattern to become less predictable. Let us look at the breadth-first dungeon and its room generation order in fig. 8.6. We can see that room 2 is connected to room 1 but room 3 is not connected to room 2. The connection of the rooms is not sequential. While room 3 does have the correct room type based on the room generation order, it does not appear to be following the order if we look at only the connection of the rooms.

8.5 Evaluation Conclusions

By looking at the modified path difference heuristic and choice heuristic scores we can say that the breadth-first dungeons are more enjoyable to play compared to the depth-first dungeons. They are more enjoyable as they have more branching paths from which the player can choose, and the type of rooms are less predictable. Furthermore, the player is not forced to visit many rooms before reaching the final room. This can be considered a flaw as the player can accidentally reach the final room without exploring the rest of the dungeon. To solve this, a key and lock system similar to the one implemented by Gellel and Sweetser can be implemented. The final room will be locked, and the keys will be scattered across the dungeon to prevent the player from progressing quickly. While the breadth-first generator is better at creating explorable dungeons, the depth-first generator can still be used to create a linear experience where the player is forced to visit many rooms before reaching the end.

CHAPTER 9

Conclusion

From our research on the procedural generation of roguelike dungeons, we observed that the procedural generation of dungeons in most cases follows the constructive approach. This is because unlike vast and open terrains, dungeons have a rigid structure that can easily be defined. The constructive approach to generating a dungeon has two major steps. The first step revolves around the spatial representation of the dungeon and its other rules such as room types, items, etc. From the generators created by Gellel et al. and Werneck et al., we observed that a grid such as a cellular automaton is commonly used to represent the space of a dungeon. The second step revolves around an algorithm that will generate the layout of the dungeon based on the spatial representation and rules presented in the first step. The generators made by the researchers mentioned previously made use of the Unity game engine to create the layout of the dungeon.

Using this information, we created two different PCG algorithms implemented in Java. One was built using breadth-first search and the other was built using the concepts of depth-first search with backtracking. The space of the dungeon was represented using a grid in the form of a 2D array. Constraints were created for the different room types and layouts of the rooms. The depth-first algorithm and breadth-first algorithm were responsible for building the layout of the dungeon based on the constraints laid out above. Finally, using Unity, a three-dimensional layout was created based on the information provided by the previous generators.

The generators were able to create complete dungeons of various sizes within a short period, 100% of the time. While the generators were able to accomplish the task they were designed to do, they are simple in nature, with room to add more complexity. In other words, now that the generator can create a dungeon, we can now add more content to the dungeon to make it come to life. One aspect that could be modified is the randomness of the room types, similar to the example of the cursed room given in section 1 of chapter 7. Furthermore, other content such as different enemies, boss enemies, items, and an objective to complete like the lock and key combination used by Gellel and Sweetser could be added.

Currently, the two-dimensional generator implemented in Java, and the Unity generator run separately. The information obtained from the two-dimensional generator must be manually given to the Unity generator. In the future, this thesis can be expanded by combining these two generators such that they work in unison, one after the other. Furthermore, if the dungeon generator is expanded to add more content as mentioned previously, the Unity generator can be turned into a fully-fledged roguelike game.

The dungeons generated by these generators were evaluated using the choice heuristic and a modified version of the path difference heuristic presented by Gellel and Sweetser. We found that breadth-first dungeons consistently performed better than depth-first dungeons based on these two metrics. The performance gap only grew as the number of rooms in the dungeon increased. The evaluation using these metrics was far from perfect. The metrics were only calculated for a few specific examples, causing a lack of observations. This evaluation can be improved by utilizing an algorithm that simulates the dungeon generators. The algorithm will increase the number of rooms generated in each iteration and calculate the choice and modified path difference heuristic for each simulation. This way we will have continuous observations that can be used for the creation of models such as a linear regression model to form a more concrete conclusion regarding the enjoyability of the generated dungeons.

In this thesis, we were successfully able to accomplish our goal of creating a procedural content generator for roguelike dungeons. Furthermore, we have created a foundation that can be expanded upon in the future.

References

[1] International Game Developers Association. "Developer Satisfaction Survey 2021". In: (2021). URL: https://igda-website.s3.us-east-2.amazonaws.com/wp-content/uploads/2021/10/18113901/IGDA-DSS-2021_SummaryReport_2021.pdf (page 1).

[2] Jordan Bell and Brett Stevens. "A survey of known results and research areas for n-queens". In: *Discrete Mathematics* 309.1 (2009), pp. 1–31. ISSN: 0012-365X. DOI: https://doi.org/10.1016/j.disc.2007.12.043. URL: https://www.sciencedirect.com/science/article/pii/S0012365X07010394 (page 39).

[3] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844 (page 34).

[4] Techie Delight. *Find ancestors of a given node in a binary tree*. URL: https://www.techiedelight.com/wp-content/uploads/Root-To-Leaf-Paths.png (page 36).

[5] Rempton Games. *How Minecraft Generates Massive Virtual Worlds from Scratch*. 2021. URL: https://remptongames.com/2021/02/28/how-minecraft-generates-massive-virtual-worlds-from-scratch/ (page 12).

[6] GeeksforGeeks. *N Queen Problem | Backtracking | GeeksforGeeks*. 2017. URL: https://www.youtube.com/watch?v=0DeznFqrgAI (page 40).

[7] Alexander Gellel and Penny Sweetser. "A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels". In: *International Conference on the Foundations of Digital Games*. FDG '20. Bugibba, Malta: Association for Computing Machinery, 2020. ISBN: 9781450388078. DOI: 10.1145/3402942.3402945. URL: https://doi.org/10.1145/3402942.3402945.

[8] Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. *Data Structures and Algorithms in Python*. 1st. Wiley Publishing, 2013. ISBN: 1118290275 (pages 33–34, 36).

[9] Fortune Business Insights. "Video Games Market Size, Share & COVID-19 Impact Analysis, By Device (Smartphones, PC/Laptop, and Consoles), By Age Group (Generation X, Generation Y, and Generation Z), By Platform Type (Online and Offline), and Regional Forecast, 2022-2029". In: (2023). URL: https://www.fortunebusinessinsights.com/video-game-market-102548 (page 1).

[10] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. "Cellular Automata for Real-Time Generation of Infinite Cave Levels". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. PCGames '10. Monterey, California: Association for Computing Machinery, 2010. ISBN: 9781450300230. DOI: 10.1145/1814256.1814266. URL: https://doi.org/10.1145/1814256.1814266 (pages 22–23).

[11] A.R. Leach. "4.05 - Ligand-Based Approaches: Core Molecular Modeling". In: *Comprehensive Medicinal Chemistry II*. Ed. by John B. Taylor and David J. Triggle. Oxford: Elsevier, 2007, pp. 87–118. ISBN: 978-0-08-045044-5. DOI: https://doi.org/10.1016/B0-08-045044-X/00246-7. URL: https://www.sciencedirect.com/science/article/pii/B008045044X002467 (page 18).

[12] The SPUF of Legend. *Procedural Generated Content in Left 4 Dead 2*. 2018. URL: `https://www.youtube.com/watch?v=81p07PsVRoU` (page 10).

[13] Richard C. Moss. "ASCII art + permadeath: The history of roguelike games". In: (2020) (pages 3, 5–6).

[14] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597 (page 39).

[15] Noor Shaker, Julian Togelius, and Mark J Nelson. *Procedural content generation in games*. Springer, 2016 (pages 7, 11, 14–16, 19–23).

[16] Steven S. Skiena. *The Algorithm Design Manual*. 2nd. Springer Publishing Company, Incorporated, 2008. ISBN: 1848000693 (page 38).

[17] Queensland University of Technology. "Dr Penny Sweetser". In: (). URL: `https://www.qut.edu.au/about/our-people/academic-profiles/penny.sweetser` (page 24).

[18] Julian Togelius et al. "What is Procedural Content Generation? Mario on the Borderline". In: *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. PCGames '11. Bordeaux, France: Association for Computing Machinery, 2011. ISBN: 9781450308724. DOI: `10.1145/2000919.2000922`. URL: `https://doi.org/10.1145/2000919.2000922`.

[19] Australian National University. "Dr Penny Kyburz". In: (). URL: `https://cecc.anu.edu.au/people/penny-kyburz` (page 24).

[20] Mariana Werneck and Esteban W. G. Clua. "Generating Procedural Dungeons Using Machine Learning Methods". In: *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. 2020, pp. 90–96. DOI: `10.1109/SBGames51465.2020.00022`.

[21] Left 4 Dead Wiki. *The Director*. URL: `https://left4dead.fandom.com/wiki/The_Director` (page 13).

[22] Wikipedia. *A procedurally generated dungeon in the 1980 version*. URL: https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/Rogue_Screenshot.png/440px-Rogue_Screenshot.png (page 7).

[23] Wikipedia. *Development of No Man's Sky*. URL: https://en.wikipedia.org/wiki/Development_of_No_Man%5C%27s_Sky#cite_note-37 (pages 12–13).

[24] Wikipedia. *Minecraft*. URL: https://en.wikipedia.org/wiki/Minecraft#cite_note-NotchExplain1-32 (page 12).

[25] Wikipedia. *Roguelike*. URL: https://en.wikipedia.org/wiki/Roguelike (page 6).

[26] WolframAlpha. *Square Grid*. URL: https://mathworld.wolfram.com/images/eps-svg/SquareGrid_800.svg (page 42).

[27] IEEE Xplore. “Esteban Clua”. In: (). URL: https://ieeexplore.ieee.org/author/37266943600 (page 29).

[28] Georgios Yannakakis and Julian Togelius. “Experience-Driven Procedural Content Generation”. In: *Affective Computing, IEEE Transactions on* 2 (July 2011), pp. 147–161. DOI: 10.1109/T-AFFC.2011.6 (pages 9–10).