Bachelor of Science in Digital Game Development
June 2025

Algorithms used for procedurally generated dungeons

A comparison between Binary Space Partitioning, Depth-First Search, 2D Delaunay Triangulation, and 3D Delaunay Triangulation.

Filip Michael

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

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

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

Contact Information:
Author(s):
Filip Michael
E-mail: fimi17@student.bth.se

University advisor:
Senior Lecturer Valeria Garro
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background: Different types of procedural generation are widely used in everyday life, from movies to video games. In video games, the primary purpose is to simplify work and automatically generate usable content with minimal user input. One common use case of procedural generation in video games is room-based dungeon generation. This study compared four different dungeon generation algorithms: Binary Space Partitioning (BSP), Depth-First Search (DFS), 2D Delaunay Triangulation (DT 2D), and 3D Delaunay Triangulation (DT 3D).
Methods: Implementation and experimentation were used to evaluate and compare the algorithms. Each algorithm was executed 10 times for each algorithm using three different settings and room sizes. The settings were changed so that the number of rooms generated was approximately 25, 50, and 100 rooms. The data generated from the tests was collected and used to identify the highest, lowest, and average values for each property of each algorithm.
Objectives: This study aimed to better understand the different algorithms' advantages and disadvantages to assess their suitability for procedural dungeon generation. To determine their suitability, several different criteria were selected. The criteria were as follows: CPU usage, execution time, RAM usage, and how hardware affects performance.
Results: The test results showed that DFS was the overall winner in all categories, followed by DT 2D, BSP, and DT 3D. DT 3D showed severe performance degradation due to high CPU and RAM usage when generating larger dungeons, making it unusable. Hardware only affected execution time and had little to no other effects on RAM or CPU usage.
Conclusions: More extensive testing in the future is needed to see how different generation setting impacts the algorithm's performance. This could give insight into how to further optimise the algorithms, making them more efficient.

Keywords: Procedural Content Generation, Binary Space Partitioning, Delaunay Triangulation, Dungeon Generation, Depth-First Search.

Acknowledgments

I would like to thank my supervisor Valeria Garro for all the help and feedback received throughout this project.

Contents

Chapter 1

Introduction

Nowadays, different types of procedural generation are widely used in everyday life, from movies to video games. The primary purpose of procedural generation is to simplify work and automatically generate usable content [15] [18]. This technique is often referred to as procedural content generation or PCG.

A prominent example of PCG in video games is the 2016 game No Man's Sky published by Hello Games [4], where PCG is heavily used to procedurally generate most of its in-game universe. Nearly every aspect of No Man's Sky is procedurally generated, except for the names of the first five galaxies. The procedurally generated content includes its planets, solar systems, ships, and creatures. This allows the game to feature over 18 quintillion unique planets. This is achieved through an underlying framework of rules and algorithms rather than manual design. This allows for a significant decrease in development time while increasing content diversity.

One of the first things that comes to mind when thinking about procedural generation is some kind of terrain generation. Typically, this involves grayscale images called heightmaps, which represent elevation in terrain generated. These images are often generated using some kind of noise-based algorithm. Two of the most common and widely used noise-based algorithms for terrain generation are Perlin noise, created by Ken Perlin and published in [9], and Simplex noise, a later refinement and improvement of Perlin noise published in [10]. When it comes to video games, procedural generation goes far beyond terrain generation. It is commonly used to create entire video game levels, such as a dungeon in a dungeon crawler. These types of dungeons are often procedurally generated; this offers the player a new and unique experience each time they play.

There exists a wide range of different algorithms for generating such content, each with its advantages and disadvantages. Some prioritise performance, while others prioritise variety or complexity. As such, choosing the right algorithm heavily depends on what requirements the user has for their specific application. This thesis aims to compare four different procedural generation algorithms used for dungeon creation: Binary Space Partitioning (BSP), Depth-First Search (DFS), 2D Delaunay Triangulation (DT 2D), and 3D Delaunay Triangulation (DT 3D). The goal is to compare their advantages and disadvantages to better understand their suitability for procedural dungeon generation.

1.1 Background

The main purpose of PCG is to simplify and accelerate the creation of content with minimal or indirect user input. While PCG can exist in various forms of content, this thesis will focus specifically on the procedural generation of dungeons consisting of interconnected rooms. There exist several different algorithms used for procedural dungeon generation; however, this study will examine four different algorithms: BSP, DFS, DT 2D, and DT 3D. These algorithms were chosen due to their prevalence in the field and the availability of extensive documentation, making them suitable for comparative analysis.

1.2 Aim and Objectives

Comparison between BSP, DFS, and Delaunay Triangulation algorithms used for procedural dungeon generation is not widely reported. This study aims to gain an understanding of the performance differences between BSP, DFS, DT 2D, and DT 3D by implementing and experimenting by gathering performance data when they are used for procedural dungeon generation in the Unity game engine. The data that will be collected is CPU usage, execution time, and RAM usage.

Objectives of this study:

- Implement Binary Space Partitioning, Depth-First Search, and 2D Delaunay Triangulation in Unity.
- Measure and compare the performance of each algorithm using defined metrics (execution time, RAM and CPU usage).
- Analyse the qualitative aspects of each algorithm, such as variety and implementation complexity.
- Based on performance and the qualitative analysis, determine use cases for each algorithm.

1.3 Research Question

RQ1 What are the quantitative performance differences (execution time, RAM and CPU usage) between Binary Space Partitioning, Depth-First Search, 2D and 3D Delaunay Triangulation when used for procedural dungeon generation?

RQ2 What qualitative differences (dungeon structure, implementation complexity et cetera) exist between Binary Space Partitioning, Depth-First Search, 2D and 3D Delaunay Triangulation algorithms when used for procedural dungeon generation?

RQ3 How does the performance of the algorithms vary between newer hardware (2022) and older hardware (2013) configurations?

1.4 Document Layout

This thesis is divided into six main chapters:

Related Work
In related work, an overview is provided of existing research relevant to procedural generation algorithms, with a focus on dungeon generation in games.

Theory
This chapter will go over the basic theory of how the algorithms work.

Method
The methodology chapter explains the selected algorithms, how they were implemented, and tested, and the criteria used for comparison.

Results and Analysis
This chapter presents the outcomes of the experiment, including visual and quantitative results, and interprets their significance.

Discussion
This chapter discussed the results from the previous chapter, including strengths, weaknesses, and possible limitations of the selected algorithms.

Conclusions and Future Work
In conclusion, a summary of the results from this study is provided, and with a conclusion to the research question and in future work, all ideas of how this research subject can be expanded upon are presented.

Chapter 2

Related Work

The study of procedural generation algorithms when used for dungeon generation has been explored in various works, though direct comparisons of algorithms are relatively few. The key reference for this thesis is the master's thesis by Oliver Karlsson [7]. Karlsson compares Binary Space Partitioning (BSP) and Delaunay Triangulation (DT) as methods for generating dungeon maps in roguelike games. While Karlsson's work shares some similarities with the focus of this thesis, there are some notable differences. Karlsson examines the execution time, variation, similarity, and the ability to connect all rooms in the dungeons. In contrast, this thesis places greater emphasis on the algorithm's overall performance metrics and not only on execution time. Karlsson's thesis concludes that BSP-generated dungeons are 25% faster than DT up to a dungeon size of 128 rooms, where the difference in performance increases drastically.

2.1 Alternative Algorithms

There exist several different algorithms that can be used for procedural dungeon generation in addition to the ones used in this thesis. Some notable examples of such algorithms are machine learning and cellular automaton Et cetera.

Werneck et al. [18] propose the use of machine learning to solve the lack of control and mischaracterization of the game design when using PCG for dungeon generation. Their user study demonstrated that their method generated reliable maps that were more enjoyable than manually generated dungeons following the same design principle.

In [6] Yahya et al. combine the use of Cellular Automata (CA) with Poisson Disk Sampling (PDS) to generate a 2D dungeon. To achieve this Yahya implemented a modified, grid-based version of PDS for increased computational efficiency and used the DFS maze generation algorithm to connect each room. Yahya concluded that using only CA was insufficient for generating dungeons with specific designs and that certain pipelines were required to be adjusted to control the results.

Putra et al. [12] combined the use of BSP with L-Systems to create game-ready dungeons. BSP is used to perform efficient space partitioning, while L-System generate diverse and visually appealing corridors between rooms resulting in a generated dungeon that has a good level of playability.

Dutra et al. [3] explore the use of Artificial Intelligence (AI) techniques, specifically reinforcement learning, to generate levels for 2D dungeon crawler games using the PCGRLPuzzle framework. Dutra generated levels for three different games using this method, concluding that utilizing reinforcement learning together with PCG enabled the generation of highly diverse levels.

Gaël et al. [5] propose a constraint-based approach for procedural dungeon generation in open-world video games. Their method utilizes a global dungeon description, along with a graph-based abstraction, to represent all feasible room configurations, thereby enabling the generation of diverse and consistent dungeon layouts. Gaël concludes that this approach supports the creation of complex, narrative-rich game environments.

2.2 DFS

In [8] Kozlova et al. provide a comparative analysis of three fundamental 2D maze algorithms: DFS, Prim's, and Recursive Definition. The study focuses on the advantages and disadvantages in terms of efficiency and path length of these algorithms. DFS, which is relevant to this thesis, is identified as one of the simplest maze generation algorithms, producing long and winding paths. These properties are valuable features in the context of dungeon generation, where similar winding paths are often desirable. Kozlova's work provides thorough theoretical explanations and pseudocode for implementing DFS, which will serve as one of the bases for the DFS implementation in this thesis.

A more practical implementation of DFS for dungeon generation can be found in a video tutorial by SilverlyBee [14], where they explain how to implement DFS in Unity. This practical implementation will also serve as one of the bases for the DFS implementation in this thesis.

2.3 BSP

Shaker's et al. work in [13] provides a comprehensive overview of several different approaches for procedural dungeon generation, including space partitioning, agent-based, and cellular automata. Of particular relevance to this thesis is Shaker's detailed exploration of the BSP algorithm. Shaker thoroughly discusses both the theory behind BSP and practical pseudocode for its implementation, highlighting its advantages in terms of dungeon navigation and manipulation. This work provides a foundational understanding of BSP, which is crucial for the analysis in this thesis.

Additionally, a practical implementation of BSP is presented by Sunny Valley Studio [11], which demonstrates how to apply the algorithm in Unity. The approach outlined in this tutorial will serve as one of the bases for the BSP implementation used in this thesis.

2.4 DT 2D

In [16] Vazgriz explores the use of 2D Delaunay Triangulation for dungeon generation, detailing both the theoretical aspects and the implementation of this algorithm. The work is significant for its application of DT 2D to procedural generation in a gaming context, explaining the underlying theory and providing pseudocode.

For a more practical understanding of DT 2D, Vazgriz offers a video tutorial [17], which demonstrates the algorithm's implementation in Unity. This implementation is based on the approach used in the 2014 game TinyKeep by Phi Dinh [2]. Vazgriz's approach outlined in this tutorial will serve as the basis of the DT 2D implementation used in this thesis.

2.5 DT 3D

Similarly to DT 2D, Vazgriz also discusses the use of DT in 3D dungeon generation in [16]. The theory and practical implementation discussed for DT 3D are based on the same principles as the 2D version but adapted to three-dimensional space. Like the 2D implementation, the practical tutorial by Vazgriz [17] provides a practical demonstration of DT 3D in Unity.

Chapter 3

Theory

3.1 DFS

DFS is a recursive backtracking algorithm that carves out maze-like dungeon paths. The algorithm uses a grid composed of cells to generate the dungeon. The grid can be of any dimension that the user wants. When a suitable grid size has been chosen, the algorithm will start by picking a random cell, in most cases, the first cell created or in the list. Then the algorithm will take a random path through the grid and go as far as possible through the unvisited cells until it has nowhere else to go. It will then start going back on its own path, looking for other paths to take, and will stop when all cells have been visited.

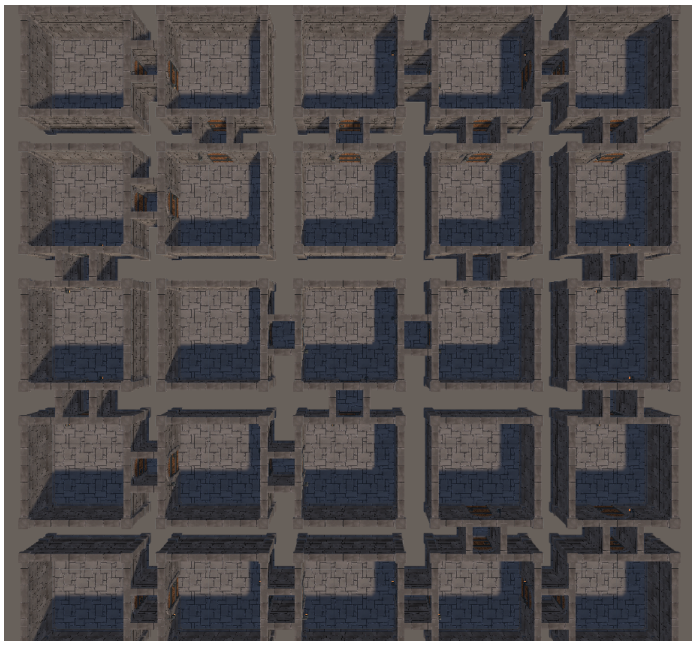

Figure 3.1: Screenshot of a 5 x 5 dungeon from above created using the DFS algorithm.

3.2 BSP

BSP is a space partitioning algorithm that recursively partitions a defined space into smaller spaces. The partitions are subdivided by either drawing a vertical or horizontal line to split them. This subdivision continues until no further subdivisions can be made, as constrained by some predefined constraint such as minimum dimensions or maximum iterations. These subdivisions are stored in what's called a BSP tree, where each node represents a partition, and each leaf node represents the final undivided region (room).

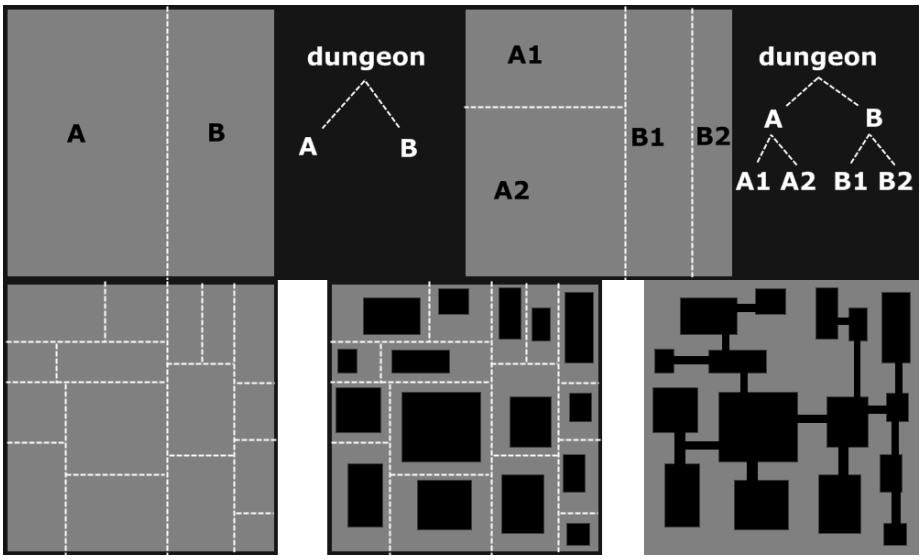

Figure 3.2: How a dungeon is created using the BSP algorithm. Adapted from images on [1]

3.3 DT 2D

A Delaunay triangulation is a mesh created from a set of points in a 2D space. This triangulation tends to avoid long and narrow triangles, resulting in a network of relatively short edges that connect to nearby points. In the context of dungeon generation, each point represents a room, with each edge representing a potential hallway between different rooms. Not every edge in the triangulation is used, as it would lead to an overall dense dungeon with too many interconnected paths. To reduce the number of edges and retain only the necessary edges for the dungeon structure. To solve this, the Bowyer–Watson algorithm is used to construct the Delaunay triangulation.

To ensure that all rooms are connected while keeping the dungeon navigable, a Minimum Spanning Tree (MST) is generated from the triangulated graph. Prim's algorithm is one method that can be used to compute the MST. This guarantees that every room is reachable; however, since it is a tree, it does not contain any cycles. Meaning that there is only a single unique path between any two rooms. However, such a dungeon may be too linear and predictable. To solve this problem,

variation and multiple paths can be introduced by randomly reintroducing some of the remaining edges not used by the MST. This makes it so that unused edges have a chance to become additional hallways. This adds cycles to the dungeon, creating a more dynamic dungeon layout.

For every hallway connection, the A* (A-Star) algorithm is used to compute the actual corridor path between the different rooms. A* will find the lowest-cost path given a graph and a cost function. The cost function is designed to favour paths through existing hallways; this encourages the pathfinder to reuse corridors instead of carving new corridors. While traversing through a room is possible, it is unlikely to happen due to how expensive it is for the pathfinder to do. Instead, the pathfinder will prefer to navigate around the rooms, resulting in short and believable hallway layouts.

3.4 DT 3D

The 3D version of Delaunay Triangulation used for dungeon generation is built upon the 2d Delaunay. Instead of triangulation, 3D Delaunay uses tetrahedralization. Unlike the 2D Delaunay, whose rooms are placed on a 2D grid, 3D Delaunay uses a 3D grid with rooms placed on different floors and uses tetrahedra to connect them and enable navigation between different floors. A minimum spanning tree (MST) is created from the tetrahedron edges, and hallways are added similarly to the 2D approach.

Pathfinding is complicated by the introduction of staircases, which must account for both vertical and horizontal movement. Staircases occupy multiple cells and require careful placement to ensure realism and usability. Hallways exist at both ends, and stairs cannot be accessed from the sides or opposite directions.

Because the staircases span multiple nodes, the A* algorithm had to be significantly modified. Nodes track their full path history to avoid reusing or overlapping staircase paths incorrectly. To make this efficient, each node uses a hash table of its path, enabling constant-time checks but requiring copying during updates, which increases complexity. The final algorithm diverges from standard A* and has an estimated time complexity of $O(N^2)$. Staircase handling is the main performance bottleneck.

Chapter 4

Method

This study uses implementation and experimentation methodology to evaluate and compare four procedural dungeon generation algorithms, DFS, BSP, DT 2D, and DT 3D. The goal is to assess the algorithms' performance, strengths, and limitations when applied to procedural dungeon generation, and how they are affected by different hardware. Implementation and experimentation were chosen as they were the most suitable options to get the required data for answering the research question that this thesis was set out to answer. Implementation, as there will be implementations of different procedural generation algorithms in Unity, and experimentation, as performance data will be collected.

4.1 Alternative Methods

While implementation and experimentation are the most suitable methodologies for this study, there are several other methodologies that could have been considered such as:

4.1.0.1 Theoretical Analysis:

This method involves analysing the algorithms using formal methods such as computational complexity or algorithmic properties. This helps identify scalability and worst-case behaviour without the need for any actual implantation of the algorithms. However, while it may help to give a theoretical insight, it does not say how the generated dungeons will perform in practice.

4.1.0.2 User Study:

A user study involves real users experiencing the generated dungeon and giving feedback on aspects such as enjoyment, difficulty or navigability. This approach helps capture human experience but can be resource-intensive, time-consuming, subjective and harder to control.

4.1.0.3 Quantitative Metrics Comparison:

This method compares algorithms based on quantifiable metrics such as connectivity, linearity or path length. These metrics can be collected from existing literature or published benchmarks, saving time, avoiding reimplementation and covering a wider range of algorithms. However, there are no guarantees that the metrics used are consistent and it might not allow for fair comparison of the different algorithms.

4.2 Tools

For the implementation and testing of the algorithms, the following was used:

- Desktop model Shark Maelstrom R510 from Shark Gaming, with Windows 11 operating system, an AMD Ryzen 5 5500 processor, 16 GB DDR4 3200 MHz RAM, and 1 TB NV2 M.2 2280 NVMe SSD.
- Desktop model ASUS M51AD from Asus, with Windows 10 operating system, an Intel i5-4440 processor, 8 GB DDR3 1600 MHz RAM, and 1 TB SATA 2.5 SSD.
- Unity game engine version 2022.3.54f1.
- Visual Studio is used to write code in Unity.
- Microsoft Excel is used to collect data and create diagrams.
- Grammarly is used for spelling and grammar.

4.3 Implementation

This chapter provides an overview of the implementation of the four algorithms. Each algorithm was implemented in C# within the Unity game engine to allow for real-time testing and visualisation. For writing the C# code, Visual Studio was used with the correct Unity plugins.

4.3.1 DFS

There are two versions of DFS used in this thesis, one for testing performance and one made to look like an actual game level. The one used for performance testing fills the whole grid so that it generates a constant number of rooms. The Unity implementation consists of two main parts: the room prefab with its corresponding script, and the generation algorithm itself that uses this room prefab.

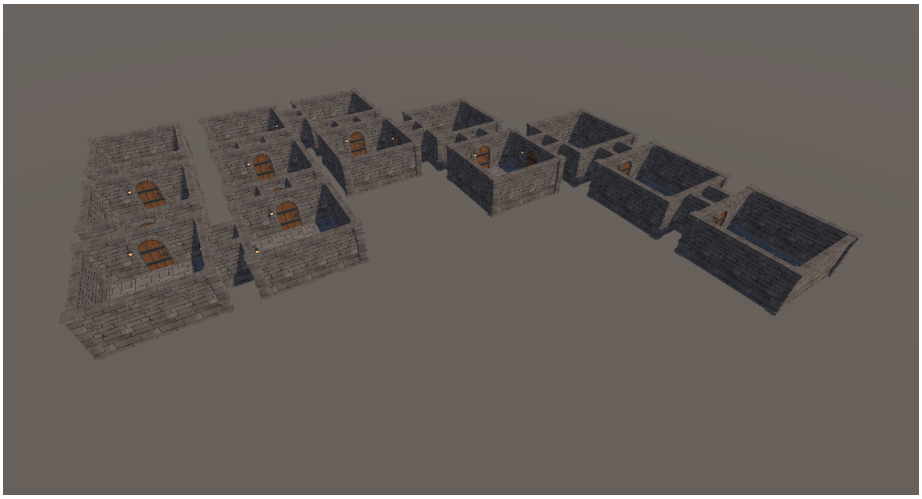

Figure 4.1: Screenshot of a dungeon created using an alternative DFS algorithm.

4.3.1.1 Room prefab

In this Unity implementation of the algorithms, the dungeon rooms are a prefab created by using a free asset package from Unity Asset Store called Stylized Hand Painted Dungeon, made by L2S ARTS. The room prefab consisted of nine walls, three on each side, with one overlapping entrance with a door on each side. Each entrance and overlapping wall are named with a specific ID to identify which direction they are facing up, down, left, and right. When generating the dungeon, this ID is used to make sure that each room has the correct entrance opened. To achieve this, the room uses a C# script to tell the prefab what entrances should be activated and what walls should be deactivated when used by the dungeon generator.

4.3.1.2 Generation

The dungeon is represented as a 2d grid made of cells. Each cell can be either a wall or a floor (walkable space). Initially, all cells are walls. The first step is to choose a starting point; this will be the first cell and will be marked as visited. Then a random direction (up, down, left, or right) will be picked. If the cells have not been visited and are not out of bounds, the wall will be knocked down between the current cell and the selected neighbour and be made a floor. The position is then moved to that neighbour. This then continues until the current cell has no unvisited neighbours that it can go to, and then it backtracks to the previous cell on the path. This goes on until it backtracks to the last cell of the path, which was our starting point.

When the path through the 2D grid of cells has been completed, each room gets instantiated at the correct position corresponding to the visited floors. When a room is placed, the room script gets called, and it checks what walls should be entrances and what walls should be walls, depending on whether the rooms have neighbours or not.

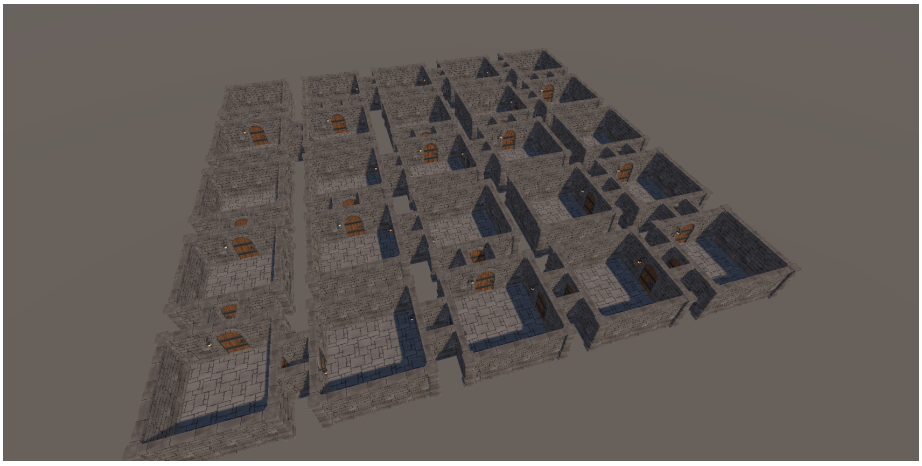

Figure 4.2: Screenshot of a 5 x 5 dungeon created using DFS algorithm.

4.3.2 BSP

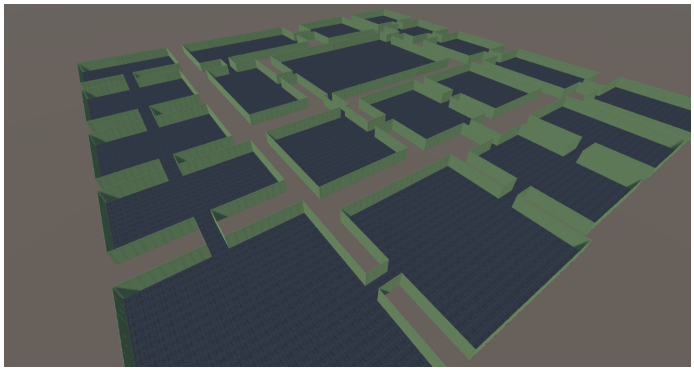

Figure 4.3: Screenshot of a 25 room dungeon created using the BSP algorithm.

4.3.2.1 Prefab

This Unity implementation of the algorithms using free assets from the Unity Asset Store is called A texture called Stone Floor Texture by ZugZug Art and Medieval Town (Base) by Kenny. The prefab used by the algorithm is the walls, and they are taken from the Medieval Town asset. The floors are not prefabs as they are created programmable using code and given the stone floor texture.

4.3.2.2 Generation

The algorithms begin by reading user-defined input parameters. These parameters are the dungeon's dimensions (width and length), minimum room dimensions, corridor width, and maximum number of partition iterations. Based on the specified dungeon dimensions given by the user, an initial rectangular area is defined. The initial area is subdivided into smaller partitions through a recursive function. The subdivisions are performed by calling a split function, which chooses a random orientation (vertical or horizontal) for each division. This continues recursively until no further subdivisions can be made, as constrained by the minimum room dimensions and maximum iterations.

For managing the hierarchical structure of the partitions, some type of data structure is needed. To solve this, a Node class was implemented, this class represents each partition in the BSP tree. This node also contains references to three other nodes, one parent node, and two child nodes, thereby forming a binary tree. Whenever a new partition is split, two new child nodes are created and assigned to the parent node representing the original partition. This process is repeated until no further subdivisions are possible.

After the BSP tree has been fully constructed, one room is added to each leaf node. Each room created is given a random size with a margin around the room. Once the rooms are added to the leaf nodes, a mesh representing the floor of each room is created. The next step involves connecting corridors to each room. This is done

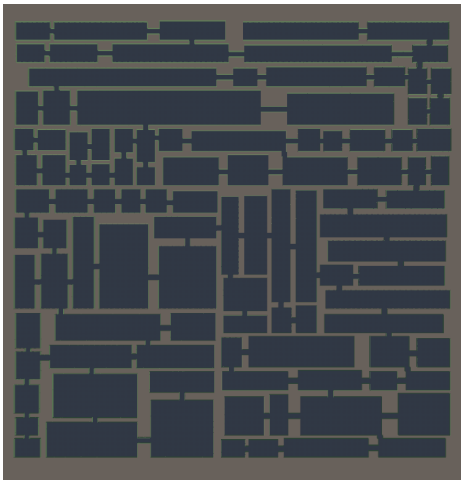

Figure 4.4: Screenshot of a 100 room dungeon from above created using the BSP algorithm.

by linking each leaf node to its sibling node. A sibling node is defined as nodes that share the same parent in the BSP tree. Once all sibling nodes have been connected, a recursive function is used to connect the sibling nodes upwards through the tree until the root node has been reached. This ensures that all rooms are accessible, and that the dungeon layout is fully connected. To visualise the corridors in 3d, floor meshes are generated, and lastly, walls are placed around the perimeter of both the rooms and corridors, completing the dungeon.

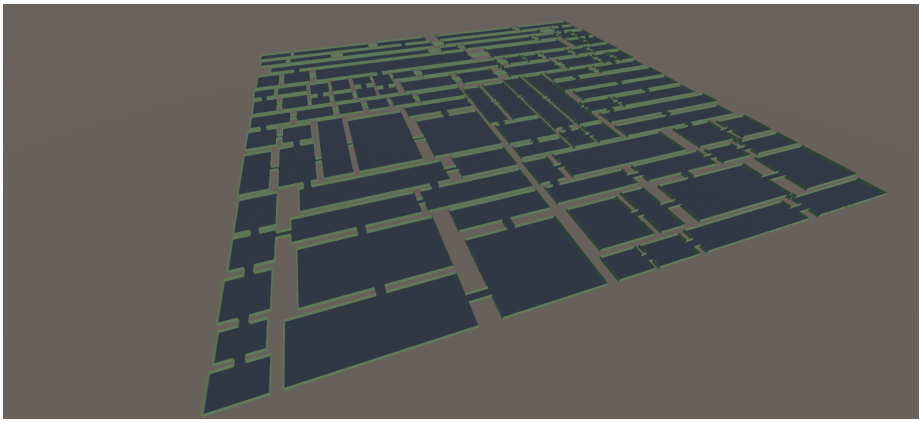

Figure 4.5: Screenshot of a 100 room dungeon created using the BSP algorithm.

4.3.3 DT 2D

4.3.3.1 Generation

The DT 2D algorithm divides the world into a grid. The rooms are given random sizes and placed randomly at non-overlapping and non-adjacent positions to ensure that they do not touch each other. Once the rooms are positioned, a Delaunay triangulation is constructed with the use of the Bowyer-Watson algorithm, with the rooms' centre treated as a vertex. This triangulation forms a graph that defines possible connections between the different rooms.

After that, a minimum spanning tree (MST) is generated from the triangulated graph using Prim's algorithm. The MST ensures that every room will be reachable, but because it is a tree, it contains no cycles. Hence, for any pair of rooms, there exists only one unique path.

To add alternative paths and improve variety, additional edges for the original triangulation graph have a random chance to be selected and added to the MST. This adds some cycles to the hallways.

For each hallway in the graph, the A* (A-star) path-finding algorithm is used to find walkable paths between the corresponding rooms. After each path is found, it modifies the world state so that future hallways can path around existing ones.

The A* algorithm uses a cost function that is used to find the lowest cost path given the graph and the cost function. This makes it cheaper to go through an existing hallway than to create a new hallway. This encourages the pathfinder to combine hallways that pass through the same area. While it is possible for a path to pass through a room, this is unlikely to happen as it is expensive. Therefore, the pathfinder will typically prefer to go around rooms instead.

4.3.4 DT 3D

This thesis will not implement DT 3D; it will instead test its performance. Instead, an already implemented algorithm created by Vazgriz will be tested. This was done because of time constraints and its complexity. It is complex enough by itself to write a paper about.

4.3.5 Data Collection

To evaluate the performance of the algorithms, Unity's built-in profiler is utilised. The focus was placed on the three metrics: CPU usage, RAM usage, and execution speed. At runtime, the profiler captures and records detailed performance data. To ensure consistency across the test runs and minimise external influences, other background processes were kept to a minimum during testing. The profiling process involves running the dungeon generation in Play Mode within the editor. The algorithm gets triggered, and then the run is stopped. The first frame in the profiler's timeline is analysed as it contains the performance data of the current algorithm's runs.

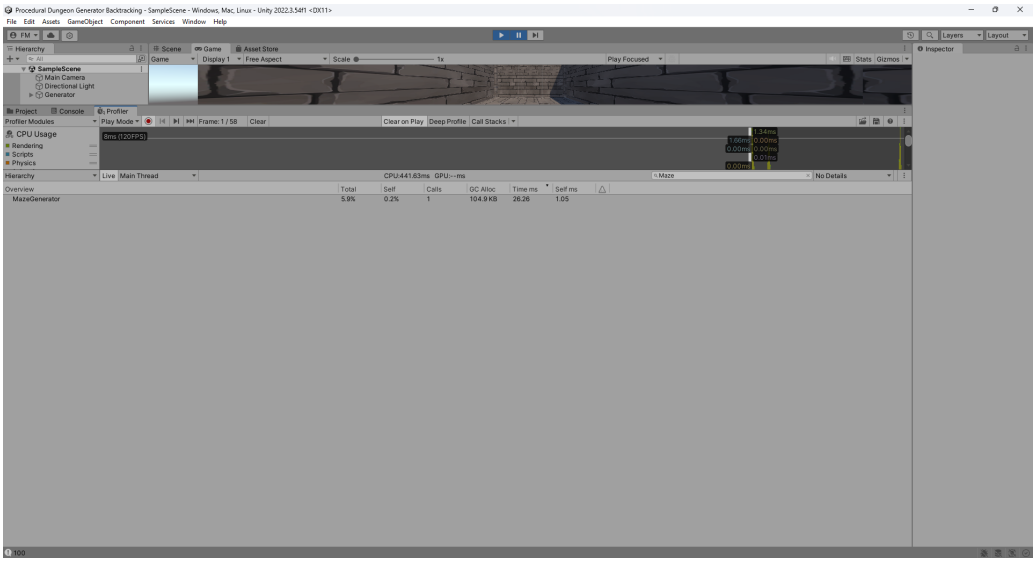

Figure 4.6: Screenshot using the profiler to measure performance.

Metrics:

1. CPU Usage: The recorded percentage of CPU usage. This metric helped assess the computational complexity of each algorithm.

2. RAM Usage: Memory usage was tracked to determine how much system memory was allocated for each algorithm during execution.

3. Execution Time (Speed): The overall time needed by each algorithm to complete its generation.

Each test was repeated multiple times to ensure reliability. The recorded data was saved to an external program, in this case, Microsoft Excel. In Excel, the data was used to create several different diagrams for each algorithm. These diagrams show the difference between test runs and the average differences between the algorithms.

4.3.6 Experiment

For this experiment, the four algorithms were performance tested in Unity game engine version 2022.3.54f1 using C#. The performance data that was measured were CPU usage, RAM usage, and execution time. The tests were conducted using two different computers, an older (2013) and a more modern (2022) one.

The newer, more modern computer used Windows 11 as its operating system, and the older computer used Windows 10. The newer computer will be called W11, and the older one will be called W10. W11 consisted of an AMD Ryzen 5 5500 CPU with a clock speed of 3.6 GHz/4.2 GHz,16 GB DDR4 RAM with a memory speed of 3200 MHz, and a 1 TB NV2 M.2 2280 NVMe SSD. W10 consisted of an Intel i5 4440 CPU with a clock speed of 3.1 GHz/3.3 GHz,8 GB DDR3 RAM with a memory speed of 1600 MHz, and a 1 TB SATA 2.5 SSD.

Each algorithm was executed 10 times for each algorithm using three different settings and room sizes. The settings were changed so that the number of rooms generated was approximately 25, 50, and 100 rooms. 25 rooms were selected as that was considered large enough to be a playable dungeon, and 100 rooms were selected as a stress test, as it was considered too large to make practical sense as a playable dungeon. The data generated from the tests was collected and stored in an Excel document, which was then used to create all the different diagrams needed for this thesis. The data was then used to identify the highest, lowest, and average values for each property of each algorithm. The data will be presented with two different types of diagrams. Box plots are also referred to as box-and-whisker diagrams and Bar charts.

4.3.6.1 Hypothesis

The hypothesis of this study is divided into several metrics:

1. CPU Usage: The CPU usage of each algorithm will significantly vary. DFS is expected to use the least CPU, followed by BSP, DT 2D, and DT 3D being the highest-consuming algorithm.

2. Execution Time: There will be a significant difference in execution time among the algorithms. DFS is expected to be the fastest, followed by BSP and DT 2D, while DT 3D is expected to be the slowest.

3. RAM Usage: DT 3D will have the highest RAM usage, followed by BSP and DT 2D, with DFS expected to use the least RAM.

4. Effect of Hardware: When it comes to hardware, it is expected to primarily affect the execution time of the algorithms with no significant effect on CPU usage or RAM usage.

There are several reasons for these assessments. When it comes to CPU usage, it comes down to each algorithm's computational complexity. DT 3D is the most complex algorithm, followed by DT 2D, BSP, and the least complex DFS. The higher the complexity of an algorithm, the more CPU usage will be needed. Execution time is significantly affected by the complexity of an algorithm; as such, DFS should be the fastest as it is the simplest of all algorithms. RAM usage is generally affected by how much data each algorithm needs to store. RAM usage does not always directly correlate with CPU usage or speed. A fast algorithm with low CPU usage can still use large amounts of RAM if it holds large data structures in RAM, and vice versa. When it comes to the effect of hardware, while newer hardware will reduce execution time, it is not expected to significantly change how much CPU or RAM an algorithm requires. These metrics are affected by the algorithm and not hardware characteristics, assuming no major bottlenecks.

4.3.7 Validity Threats

Several different threats to validity have been identified. These threats could have a potential impact on the results of this study. Not every implementation of the algorithms tested has the same level of optimization and as such may affect the fairness of the results. Unity's built-in profiler can introduce extra overhead when profiling the algorithms leading to the algorithms using extra performance which could skew the results. Unity's internal resource management such as garbage collection, background process et cetera may also affect the performance data. These factors have been taken into account when analysing and interpreting the findings of this study.

4.3.8 Validity and Reliability

To ensure the reliability and validity of the experiment, several measures were taken. The experiments were conducted using two different computers with varying hardware configurations. This was to assess whether hardware differences would significantly affect the results. This would determine whether discrepancies were limited to aspects like execution time, which could be hardware dependent, rather than other factors such as RAM usage, which should remain consistent on most hardware. Initially, each algorithm was to be executed only five times for each algorithm, using three different configurations and room sizes. However, this was increased to ten executions per algorithm to improve statistical robustness and reduce variability caused by random chance as the only algorithm that guarantees an exact number of rooms is DFS. The other algorithms' room count can vary by +-5 rooms. Additionally, external influences were minimised during testing by reducing background processes. This helped ensure that the results were not skewed by unrelated programs taking away processing power.

4.3.9 Ethical, Social, and Sustainability Aspects

This study is not expected to raise any ethical, societal or sustainability concerns. The study does not involve any human participant or collection of personal data. The only data that will be collected consists of performance data generated by the algorithms themselves. Therefore, ethical approval is not expected to be required or needed.

A potential minor societal impact could arise from the findings of this study. This study compares different procedural generation algorithms, and those results could assist developers in selecting more efficient algorithms, leading to a decrease in both development time and monetary expenses. Consequently, this may also contribute to a reduction in energy consumption, particularly when compared to doing the work manually.

Chapter 5

Results and Analysis

In this chapter, the results from the tests will be presented. First comes the results, then an analysis of why said result look like they did, and lastly, the conclusion will be drawn based on the analysis.

5.1 Data Collection

5.1.1 W11

5.1.1.1 CPU usage

The box plot below shows a comparison of the algorithms' CPU usage that was collected during the tests. Below the diagram, a table will show the minimum, maximum, and average CPU usage. The CPU usage is measured in percentages (%). For more accurate data, see Appendix Table A.1.

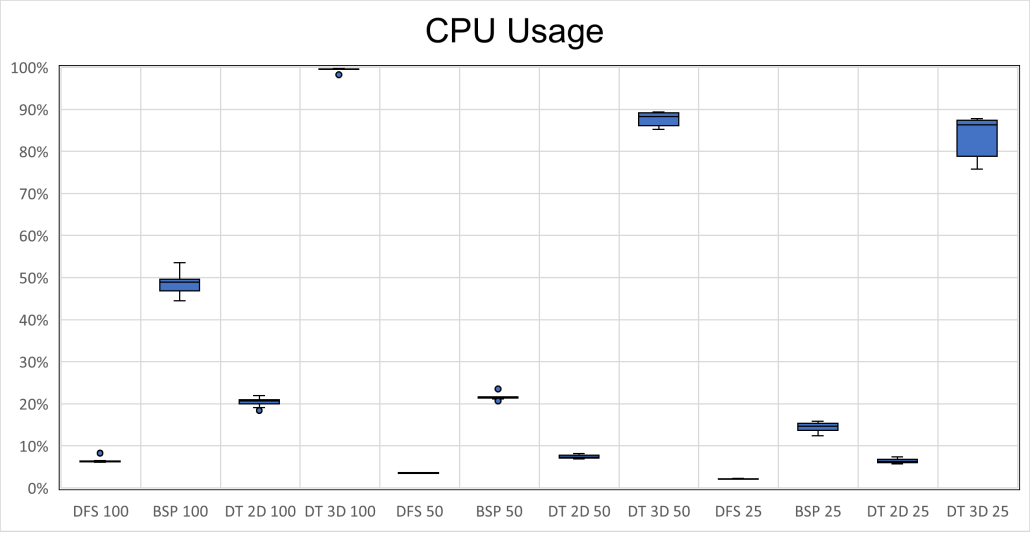

Figure 5.1: Box plot showing an overview of CPU usage for all algorithms using W11 hardware.

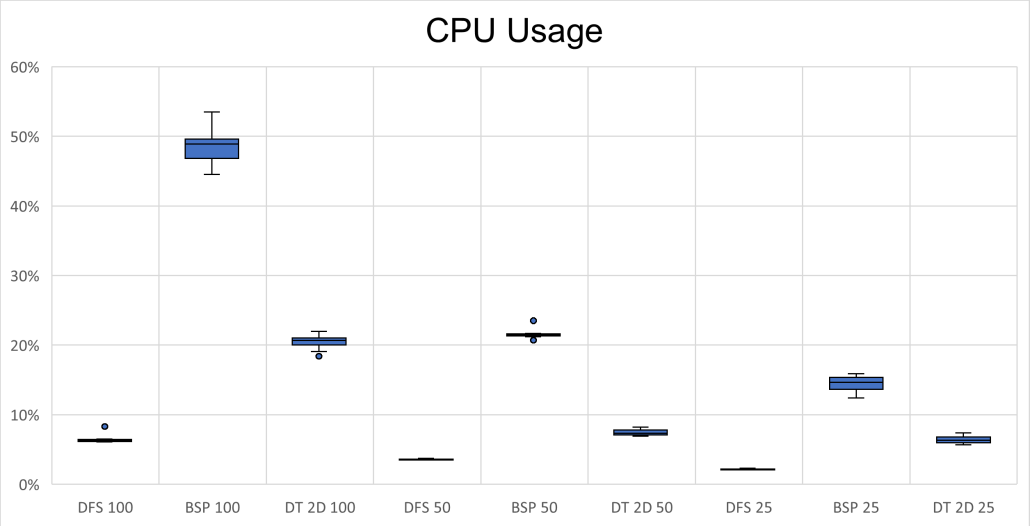

Figure 5.2: Zoomed in box plot showing an overview of CPU usage for all algorithms using W11 hardware.

Algorithm	**Room Size**	**Lowest Usage**	**Highest Usage**	**Average Usage**
DFS	100.00	**6.10%**	**8.30%**	**6.50%**
BSP	95.20	44.50%	53.50%	48.61%
DT 2D	103.90	18.40%	22.00%	20.48%
DT 3D	103.70	98.20%	99.70%	99.45%
DFS	50.00	**3.50%**	**3.70%**	**3.58%**
BSP	56.60	20.70%	23.50%	21.62%
DT 2D	48.60	6.90%	8.20%	7.44%
DT 3D	50.60	85.20%	89.40%	87.76%
DFS	25.00	**2.10%**	**2.30%**	**2.17%**
BSP	22.20	12.40%	15.90%	14.52%
DT 2D	26.00	5.70%	7.40%	6.42%
DT 3D	26.60	75.80%	87.80%	83.79%

Table 5.1: Table showing an overview of CPU usage (rounded to two decimals) for all algorithms.

5.1.1.2 Execution Time

The box plot below shows a comparison of the algorithms' execution time data that was collected during the tests. Below the diagram, a table will show the slowest, fastest, and average execution time. The execution time is measured in seconds (s). For more accurate data, see Appendix Table A.1.

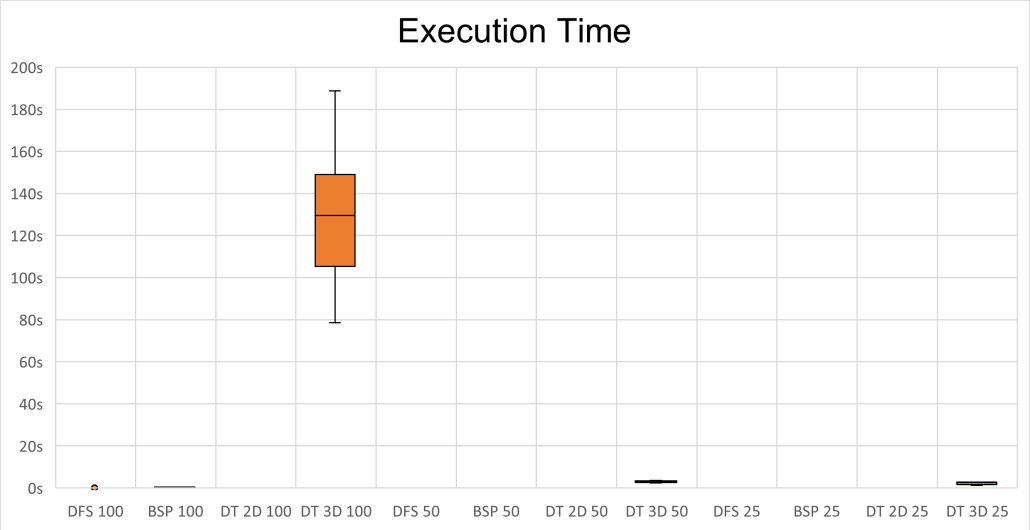

Figure 5.3: Box plot showing an overview of the execution time for all algorithms using W11 hardware.

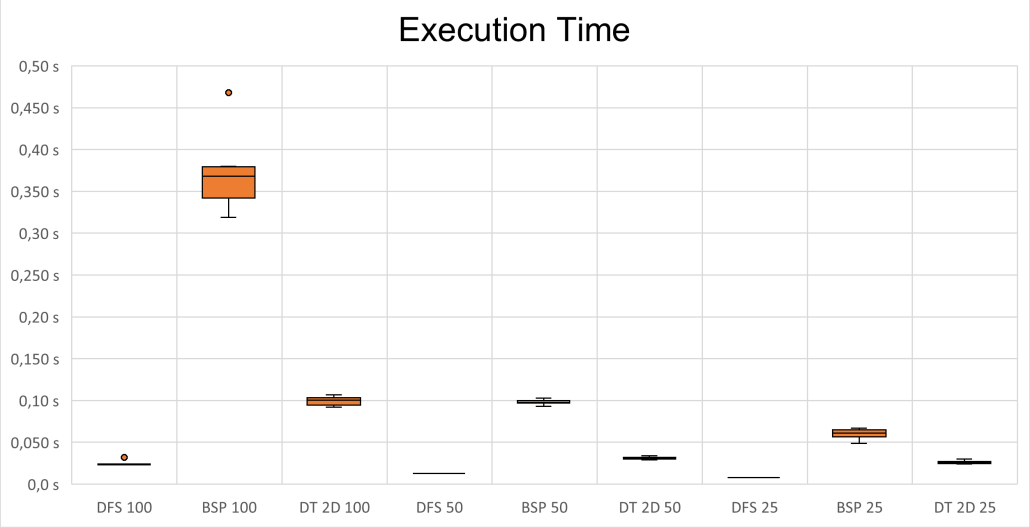

Figure 5.4: Zoomed in box plot showing an overview of the execution time for all algorithms using W11 hardware.

Algorithm	Room Size	Slowest Time	Fastest Time	Average Time
DFS	100.000	**0.032s**	**0.023s**	**0.024s**
BSP	95.200	0.468s	0.319s	0.369s
DT 2D	103.900	0.107s	0.092s	0.1s
DT 3D	103.700	188.839s	78.596s	128s
DFS	50.000	**0.013s**	**0.013s**	**0.013s**
BSP	56.600	0.103s	0.093s	0.098s
DT 2D	48.600	0.034s	0.029s	0.031s
DT 3D	50.600	3.492s	2.446s	3.018s
DFS	25.000	**0.008s**	**0.008s**	**0.008s**
BSP	22.200	0.067s	0.049s	0.06s
DT 2D	26.000	0.030s	0.024s	0.026s
DT 3D	26.600	2.750s	1.182s	2.175s

Table 5.2: Table showing an overview of the execution time (rounded to three decimals) for all algorithms using W11 hardware.

5.1.1.3 RAM Usage

The box plot below shows a comparison of the algorithms' RAM usage data that was collected during the tests. Below the diagram, a table will show the minimum, maximum, and average RAM usage. RAM usage is measured in megabytes (MB). For more accurate data, see Appendix Table A.1.

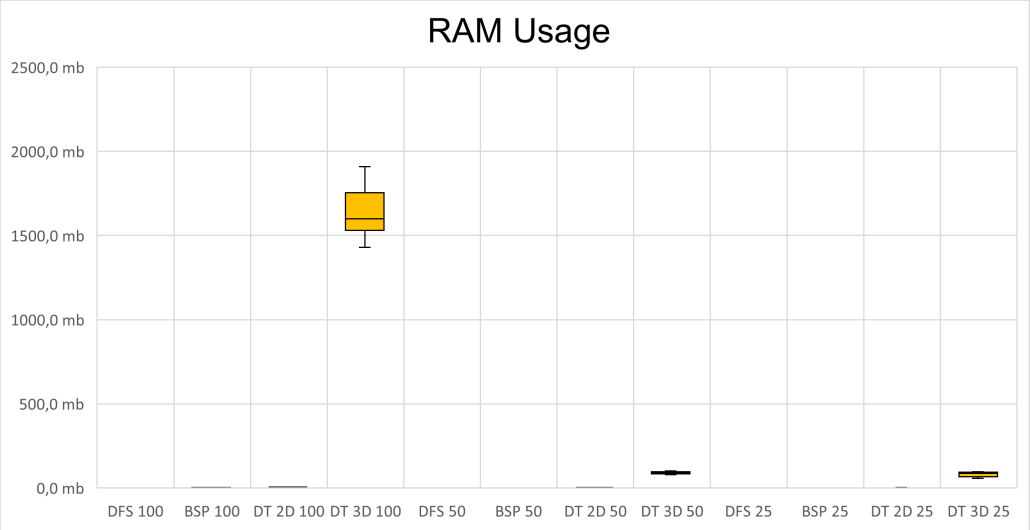

Figure 5.5: Box plot showing an overview of RAM usage for all algorithms using W11 hardware using W11 hardware.

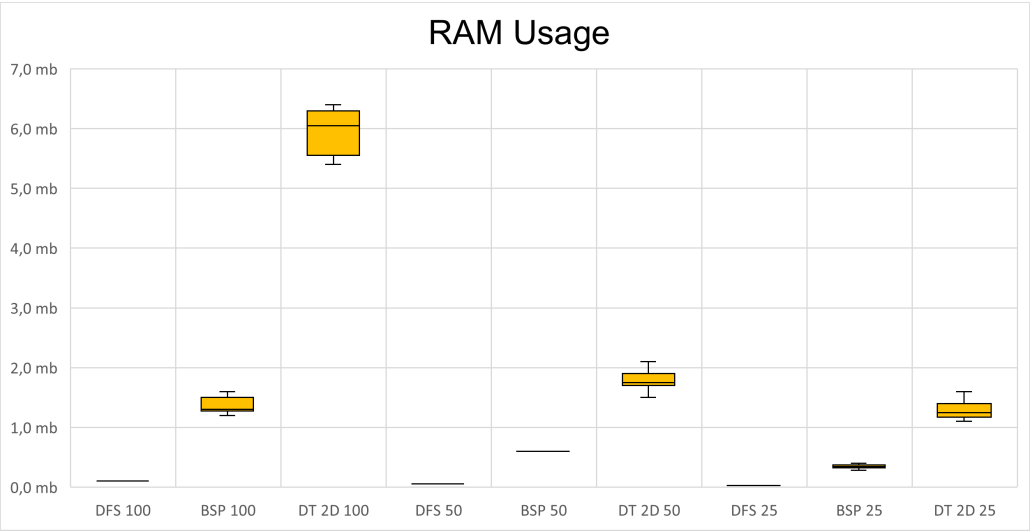

Figure 5.6: Zoomed in box plot showing an overview of RAM usage for all algorithms using W11 hardware.

Algorithm	Room Size	Lowest Usage	Highest Usage	Average Usage
DFS	100.000	**0.105 MB**	**0.105 MB**	**0.105 MB**
BSP	95.200	1.200 MB	1.500 MB	1.350 MB
DT 2D	103.900	5.400 MB	6.400 MB	5.930 MB
DT 3D	103.700	1430.000 MB	1910.000 MB	1635.000 MB
DFS	50.000	**0.053 MB**	**0.053 MB**	**0.053 MB**
BSP	56.600	0.600 MB	0.600 MB	0.600 MB
DT 2D	48.600	1.500 MB	2.100 MB	1.780 MB
DT 3D	50.600	78.900 MB	100.500 MB	91.100 MB
DFS	25.000	**0.027 MB**	**0.027 MB**	**0.027 MB**
BSP	22.200	0.285 MB	0.398 MB	0.345 MB
DT 2D	26.000	1.100 MB	1.600 MB	1.290 MB
DT 3D	26.600	57.900 MB	97.100 MB	81.610 MB

Table 5.3: Table showing an overview of RAM usage (rounded to three decimals) for all algorithms using W11 hardware.

5.1.2 W10 Hardware

5.1.2.1 CPU Usage

The box plot below shows a comparison of the algorithms' CPU usage that was collected during the tests. Below the diagram, a table will show the minimum, maximum, and average CPU usage. The CPU usage is measured in percentages (%). For more accurate data, see Appendix Table B.1.

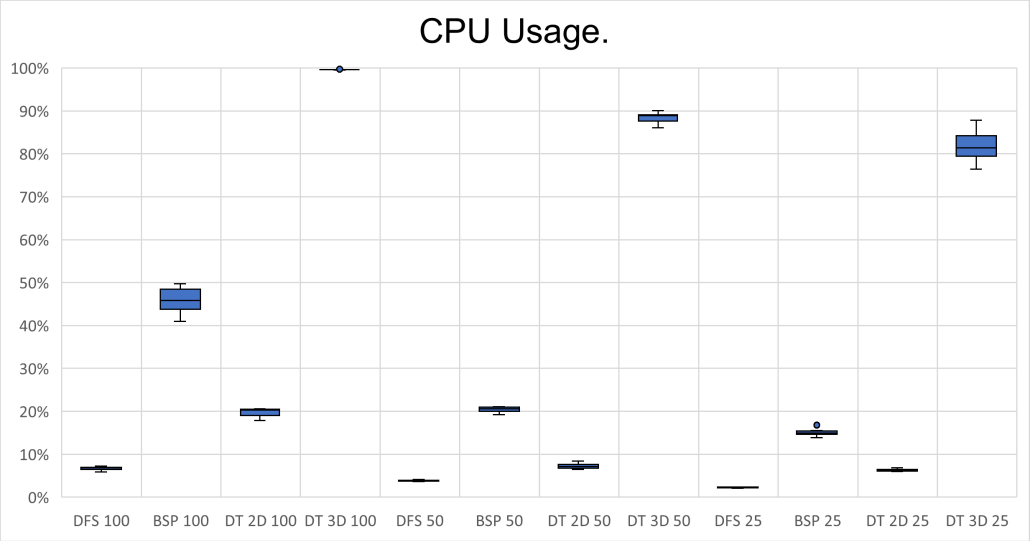

Figure 5.7: Box plot showing an overview of CPU usage for all algorithms using W10 hardware.

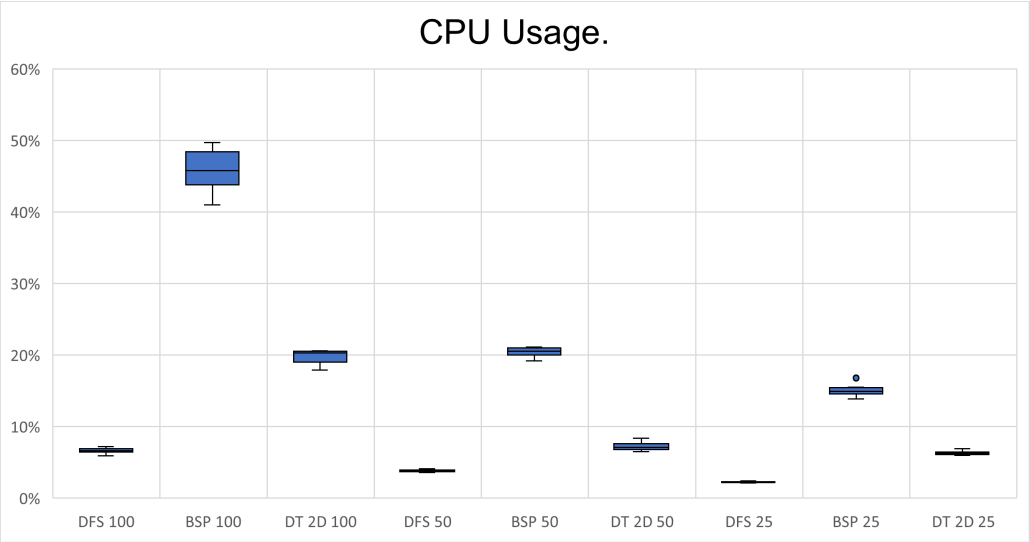

Figure 5.8: Zoomed in box plot showing an overview of CPU usage for all algorithms using W10 hardware.

Algorithm	Room Size	Lowest Usage	Highest Usage	Average Usage
DFS	100.00	**5.90%**	**7.20%**	**6.50%**
BSP	92.60	41.00%	49.70%	45.89%
DT 2D	106.70	17.90%	20.60%	19.76%
DT 3D	105.70	99.50%	99.70%	99.60%
DFS	50.00	**3.70%**	**4.10%**	**3.38%**
BSP	53.60	19.20%	21.10%	20.46%
DT 2D	49.90	6.50%	8.40%	7.26%
DT 3D	49.10	86.10%	90.10%	88.44%
DFS	25.00	**2.10%**	**2.40%**	**2.25%**
BSP	23.70	13.90%	16.80%	15.06%
DT 2D	25.90	6.00%	6.90%	6.31%
DT 3D	27.70	76.40%	87.80%	81.80%

Table 5.4: Table showing an overview of CPU usage (rounded to two decimals) for all algorithms using W10 hardware.

5.1.2.2 Execution Time

The box plot below shows a comparison of the algorithms' execution time data that was collected during the tests. Below the diagram, a table will show the slowest, fastest, and average execution time. The execution time is measured in seconds (s). For more accurate data, see Appendix Table B.1.

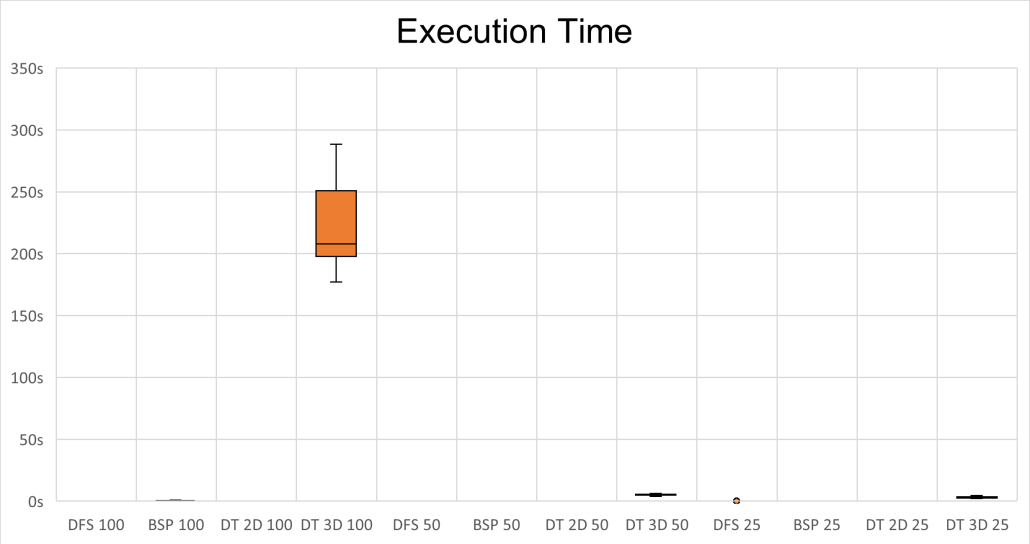

Figure 5.9: Box plot showing an overview of the execution time for all algorithms using W10 hardware.

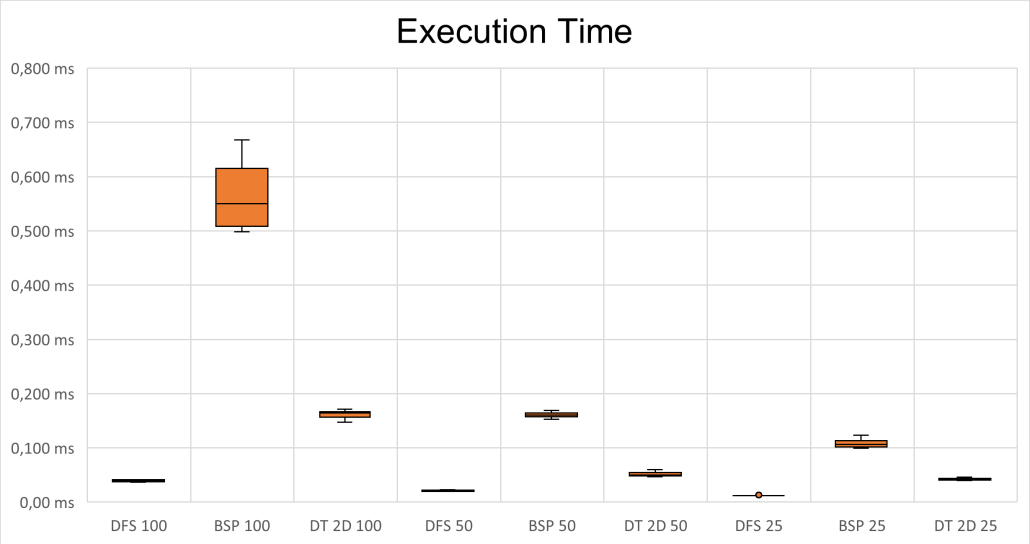

Figure 5.10: Zoomed in box plot showing an overview of the execution time for all algorithms using W10 hardware.

Algorithm	Room Size	Slowest Time	Fastest Time	Average Time
DFS	100.000	**0.041s**	**0.037s**	**0.039s**
BSP	92.600	0.668s	0.498s	0.564s
DT 2D	106.700	0.171s	0.147s	0.162s
DT 3D	105.700	288.548s	177.258s	222.772s
DFS	50.000	**0.023s**	**0.012s**	**0.021s**
BSP	53.600	0.169s	0.153s	0.160s
DT 2D	49.900	0.060s	0.047s	0.051s
DT 3D	49.100	6.077s	3.918s	5.111s
DFS	25.000	**0.001s**	**0.001s**	**0.001s**
BSP	23.700	0.123s	0.099s	0.108s
DT 2D	25.900	0.046s	0.040s	0.042s
DT 3D	27.700	4.665s	2.136s	3.009s

Table 5.5: Table showing an overview of the execution time (rounded to three decimals) for all algorithms using W10 hardware.

5.1.2.3 RAM Usage

The box plot below shows a comparison of the algorithms' RAM usage data that was collected during the tests. Below the diagram, a table will show the minimum, maximum, and average RAM usage. RAM usage is measured in megabytes (MB). For more accurate data, see Appendix Table B.1.

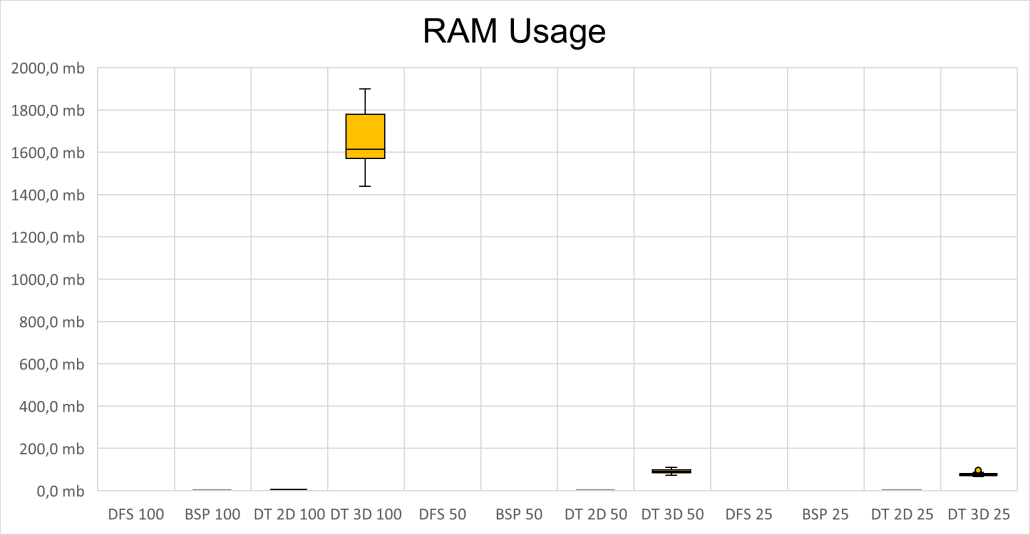

Figure 5.11: Box plot showing an overview of RAM usage for all algorithms using W10 hardware.

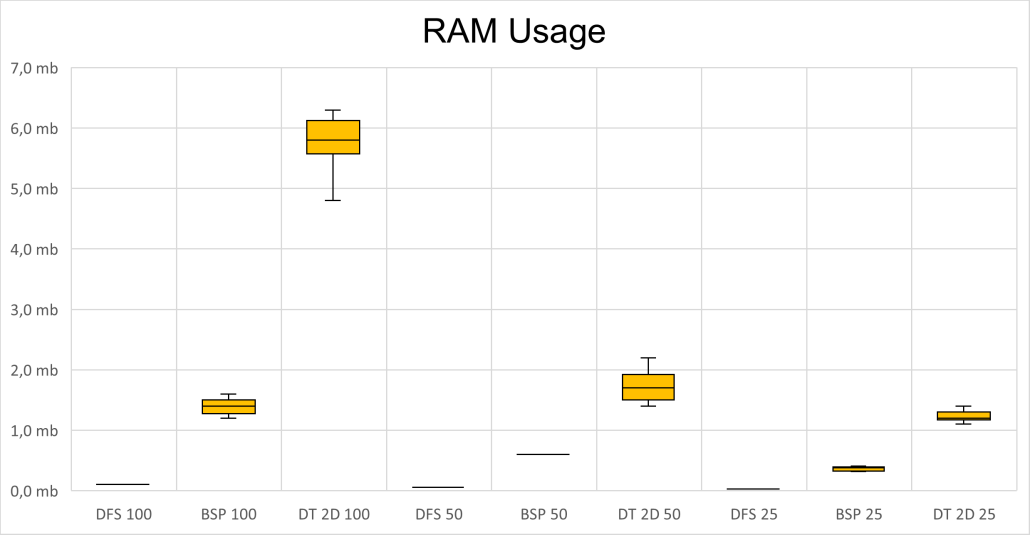

Figure 5.12: Zoomed in box plot showing an overview of RAM usage for all algorithms using W10 hardware.

Algorithm	Room Size	Lowest Usage	Highest Usage	Average Usage
DFS	100.000	**0.105 MB**	**0.105 MB**	**0.105 MB**
BSP	92.600	1.200 MB	1.500 MB	1.390 MB
DT 2D	106.700	4.800 MB	6.300 MB	5.790 MB
DT 3D	105.700	1440.000 MB	1900.000 MB	1650.000 MB
DFS	50.000	**0.053 MB**	**0.053 MB**	**0.053 MB**
BSP	53.600	0.600 MB	0.600 MB	0.600 MB
DT 2D	49.900	1.400 MB	2.200 MB	1.730 MB
DT 3D	49.100	72.700 MB	109.700 MB	90.730 MB
DFS	25.000	**0.027 MB**	**0.027 MB**	**0.027 MB**
BSP	23.700	0.316 MB	0.409 MB	0.366 MB
DT 2D	25.900	1.100 MB	1.400 MB	1.220 MB
DT 3D	27.700	70.400 MB	96.700 MB	77.070 MB

Table 5.6: Table showing an overview of RAM usage (rounded to three decimals) for all algorithms using W10 hardware.

5.2 Analysis

In this chapter, an analysis of the data collected during the tests will be presented. The analysis will compare the different properties of the algorithms and showcase the differences between them. How the results were affected by the different hardware will also be discussed.

5.2.1 CPU Usage

When comparing CPU usage with the W11 and W10 hardware, you can see that they are very similar to each other, and there is only a negligible difference. As they are within one per cent of each other. In both cases, DT 3D had the highest CPU usage by far out of all the algorithms, with more than double compared to any other algorithm. The second highest usage was BSP, third was DT 2D, with less than half the usage of BSP, and with the least usage was DFS. This trend persisted when generating all the different dungeon sizes.

5.2.1.1 W11

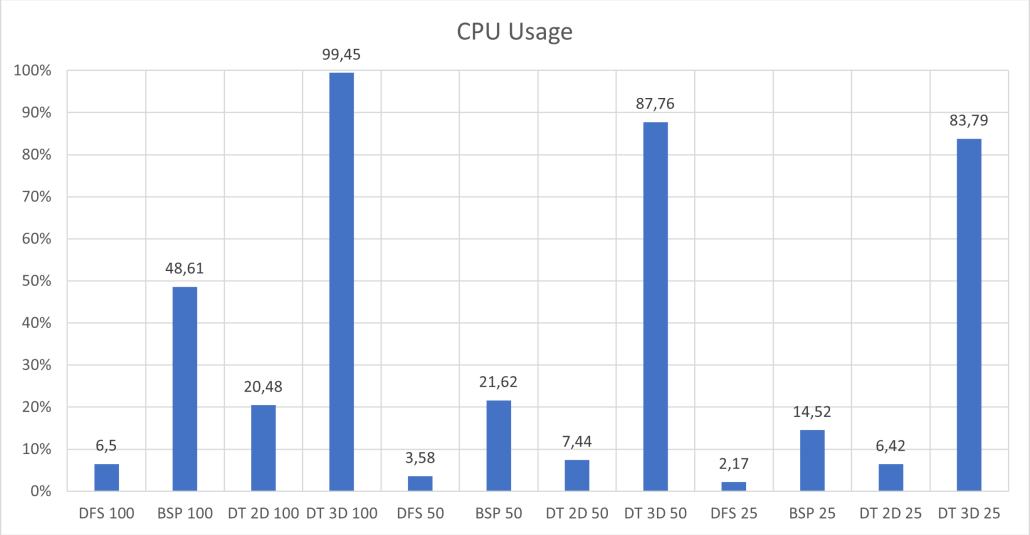

Figure 5.13: Bar chart showing an overview of CPU usage for all algorithms using W11 hardware.

As seen in the Figure, the CPU usage between different dungeon sizes changes relatively proportionally, except for DT 3D. It only changes 5 to 10 per cent between different sizes and maxes out the CPU at 100 rooms. When going from 100 to 50, the CPU usage decreased between 45 and 64 per cent for DFS, BSP, and DT 2D, and when going from 50 to 25, room CPU usage decreased between 14 and 39 per cent. For more accurate data, see the Table 5.7 below.

Comparison	CPU(%)	Speed(%)	RAM (%)
DFS 50 vs DFS 100	-45%	85%	98%
DFS 25 vs DFS 50	-39%	63%	96%
BSP 50 vs BSP 100	-56%	277%	125%
BSP 25 vs BSP 50	-33%	63%	74%
DT 2D 50 vs DT 2D 100	-64%	223%	233%
DT 2D 25 vs DT 2D 50	-14%	19%	38%
DT 3D 50 vs DT 3D 100	-12%	4141%	1695%
DT 3D 25 vs DT 3D 50	-5%	39%	12%

Table 5.7: Table showing percentage difference in performance of each algorithms.

5.2.1.2 W10

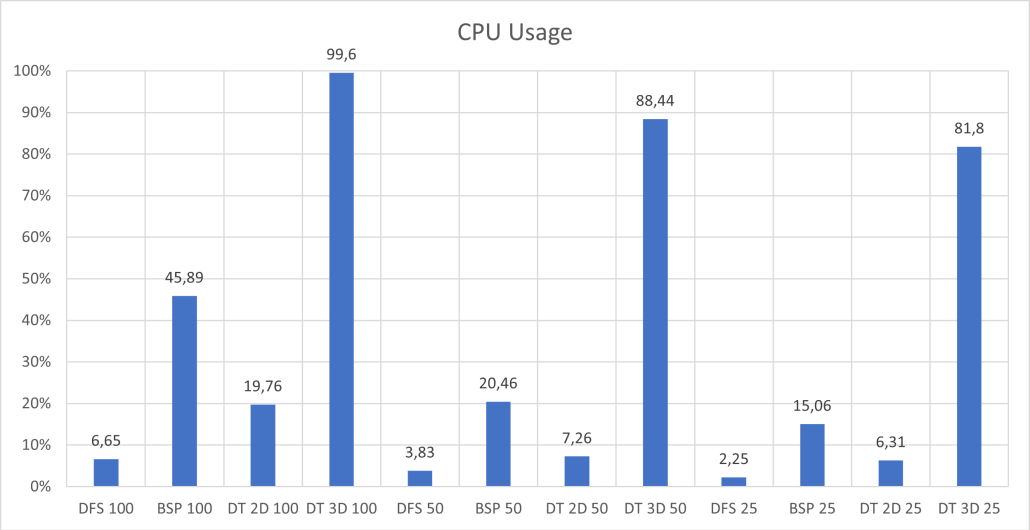

Figure 5.14: Bar chart showing an overview of CPU usage for all algorithms.

As seen in the Figure, the CPU usage between different dungeon sizes changes relatively proportionally, except for DT 3D. It only changes 8 to 11 per cent between different sizes and maxes out the CPU at 100 rooms. When going from 100 to 50, the CPU usage decreased between 42 and 63 per cent for DFS, BSP, and DT 2D, and when going from 50 to 25, the room CPU usage decreased between 13 and 41 per cent. For more accurate data, see the Table 5.8 below.

The results show that when doubling the number of rooms generated by DFS, BSP, and DT, 2D increases their CPU usage by approximately 50 per cent. When it comes to DT 3D, it only increases by 10 per cent when doubling the rooms from 25 to 50 and reaches max utilisation at 100 rooms. These results show that DFS will always be the fastest, followed by DT 2D, BSP, and DT 3D will always be the most CPU-intensive algorithm as it was the only algorithm that managed to max out the CPU utilisation.

Comparison	CPU (%)	Speed (s)	RAM (MB)
DFS 50 vs DFS 100	-42%	86%	98%
DFS 25 vs DFS 50	-41%	75%	96%
BSP 50 vs BSP 100	-55%	253%	132%
BSP 25 vs BSP 50	-26%	48%	64%
DT 2D 50 vs DT 2D 100	-63%	218%	235%
DT 2D 25 vs DT 2D 50	-13%	21%	42%
DT 3D 50 vs DT 3D 100	-11%	4259%	1719%
DT 3D 25 vs DT 3D 50	-8%	70%	18%

Table 5.8: Table showing percentage difference in performance of each algorithms.

The fact that the results look like they do is not surprising when taking into consideration the algorithms' different complexities. DT 3D is the most complex of the algorithms, with O(n2), followed by DT 2D and BSP with a complexity of O(n log n), and DFS, which is the least complex at O(v + E). This means that DT 3D should be the most CPU-intensive algorithm in most cases, followed by DT 2D, BSP, and DFS. What is more surprising is that BSP has the same complexity as DT 2D and has more than double the CPU usage in all tested cases. There can be several reasons for that, but in this case is most likely down to how the algorithm is implemented, as it is not as optimised as much as it could be.

5.2.2 Execution Time

When comparing the execution time between W11 and W10's hardware, one may see that they appear similar, but there are noticeable differences in performance. The newer hardware appears to be at a minimum of 50% faster than the older hardware for all algorithms, and this is consistent over all dungeon sizes.

5.2.2.1 W11

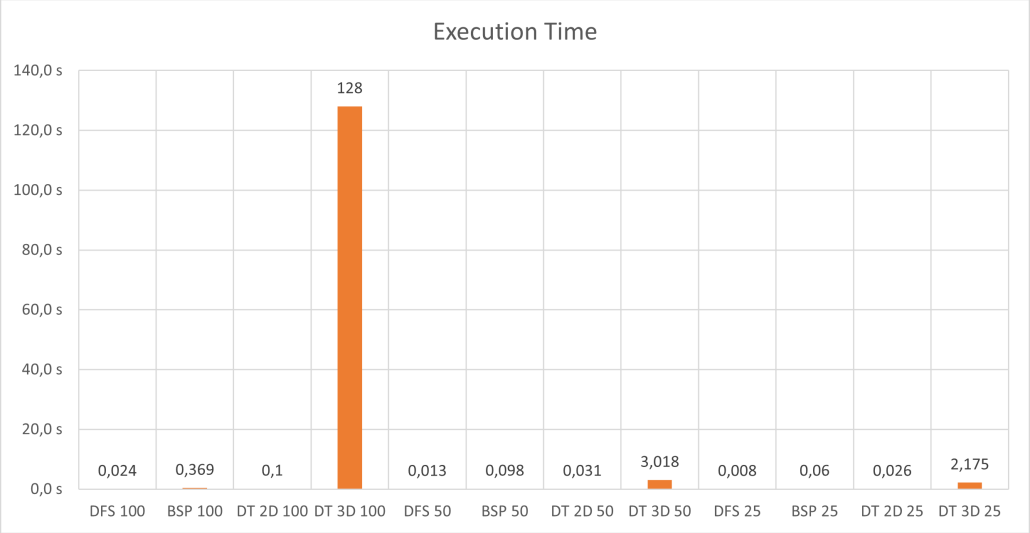

Figure 5.15: Bar chart showing an overview of the execution time for all algorithms using W11 hardware.

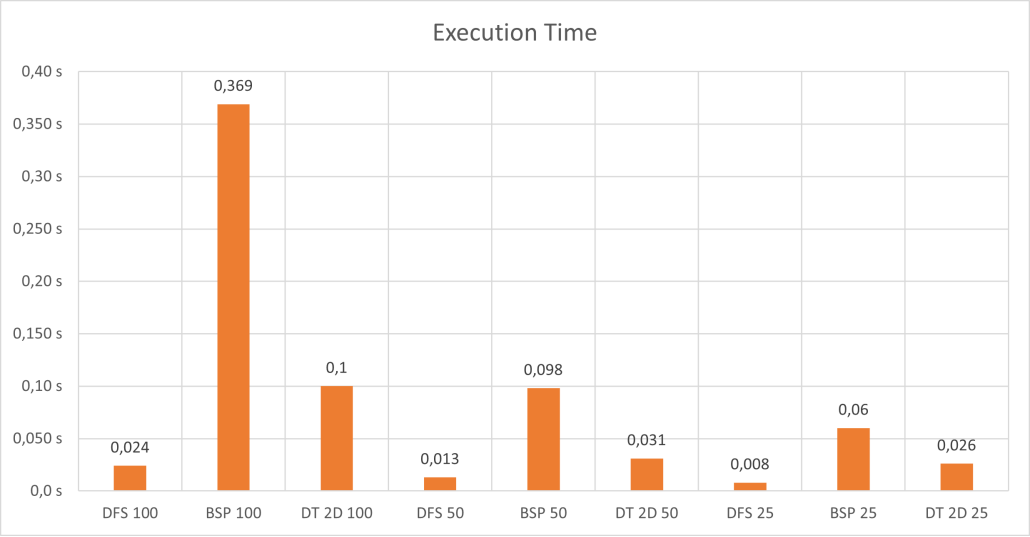

Figure 5.16: Bar chart showing an overview of the execution time for all algorithms using W11 hardware.

DFS was the fastest algorithm, followed by DT 2D, BSP placed third, and DT 3D was the slowest with a significant margin. This trend persisted when generating all the different dungeon sizes. DT 3D had by far the slowest execution time out of all the algorithms. At 100 rooms, it took over 2 minutes to generate, which is over 5000 times slower than the fastest algorithm, and in the best case, it is nearly 350 times slower than BSP. At 50 rooms and lower, the performance of DT 3D improved considerably as it took a little over three seconds to generate a dungeon, which is still slower than the other algorithms, but an enormous improvement. When generation 100, 50, and 25 rooms, all other algorithms took less than a second in all cases.

5.2.2.2 W10

DFS was the fastest algorithm, followed by DT 2D, BSP placed third, and DT 3D was the slowest with a significant margin. This trend persisted when generating all the different dungeon sizes. DT 3D had by far the slowest execution time out of all the algorithms. At 100 rooms, it took nearly 4 minutes to generate, which is over 5000 times slower than the fastest algorithm, and in the best case, it was nearly 400 times slower than the BSP. At 50 rooms and lower, the performance of DT 3D improved considerably as it took a little over 5 seconds to generate a dungeon, which is still slower than the other algorithms, but an enormous improvement. When generation 100, 50, and 25 rooms, all other algorithms took less than a second in all cases.

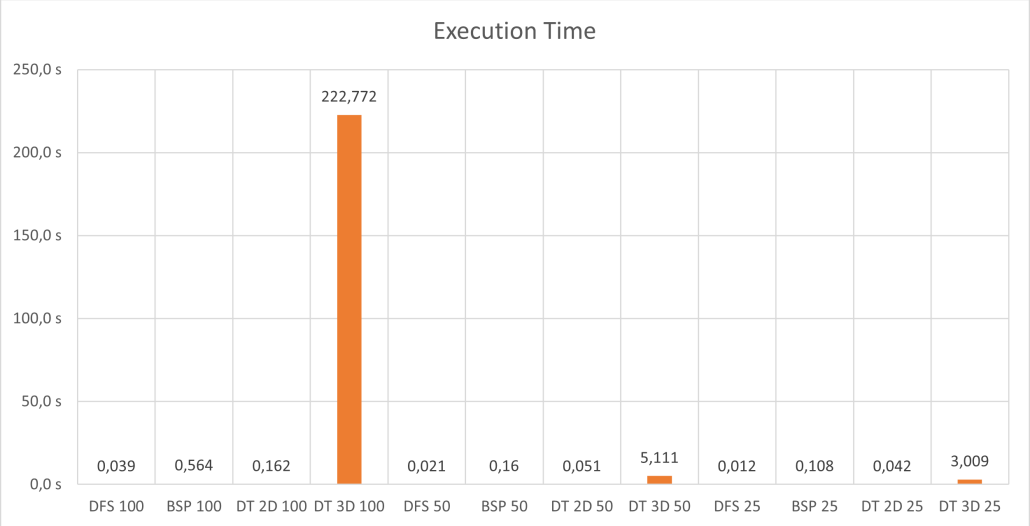

Figure 5.17: Bar chart showing an overview of the execution time for all algorithms using W10 hardware.

Figure 5.18: Zoomed in bar chart showing an overview of the execution time for all algorithms using W10 hardware.

As stated earlier, the results are mostly in line with the algorithms' complexities and, for the most part, proportional to size decrees and increases. When it comes to the execution time for DT 3D on 100 rooms, it should not be that slow when compared to the other algorithms. There may be several reasons for that. It could be that the settings used for generating 100 rooms were incompatible, implemented/optimisation, or it could also be that the algorithm is not suited for generation over 50 rooms, as it may not scale well with higher numbers.

5.2.3 RAM usage

When comparing RAM usage between W11 and W10's hardware, you can see that they are very similar to each other and there is only a negligible difference, as they are within a few per cent at most for all algorithms. In both cases, DT 3D had the highest RAM usage by far out of all the algorithms, with more than 15 times higher in the best case compared to any other algorithm. The second highest usage was DT 2D, the third was BSP, and with least usage was DFS. This trend persisted when generating all the different dungeon sizes.

5.2.3.1 W11

As seen in the Figure, the RAM usage between different dungeon sizes changes relatively proportionally, except for DT 3D. As the RAM usage increases by over 4000% when going from 50 to 100 rooms. Between 25 to 50 rooms, it increases by about 12 per cent. When going from 100 to 50 rooms, the RAM usage decreased by about 50 per cent for DFS, BSP, and DT 2D, and going from 50 to 25 rooms RAM usage decreased by about 50 per cent again. For more accurate data, see the Table 5.7 above.

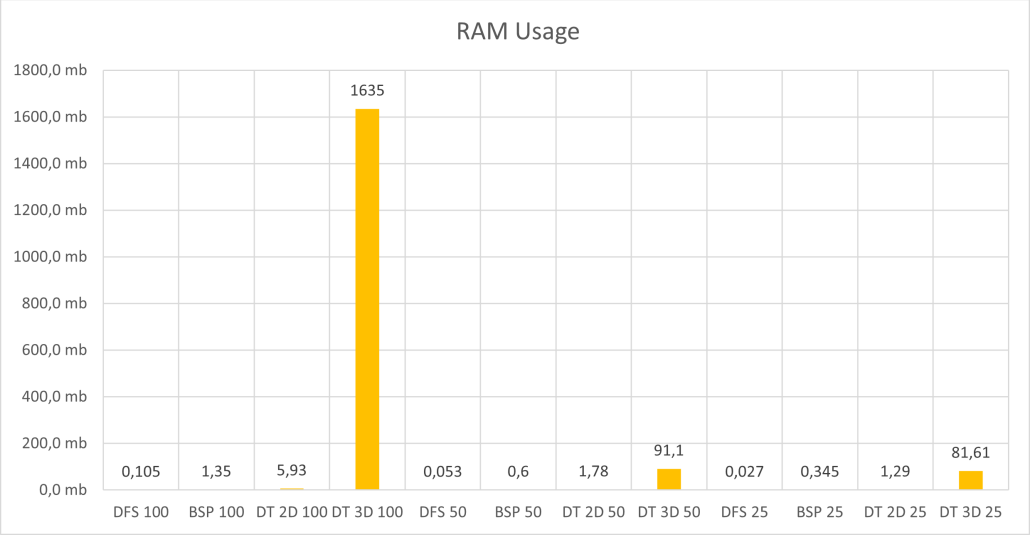

Figure 5.19: Bar chart showing an overview of RAM usage for all algorithms using W11 hardware.

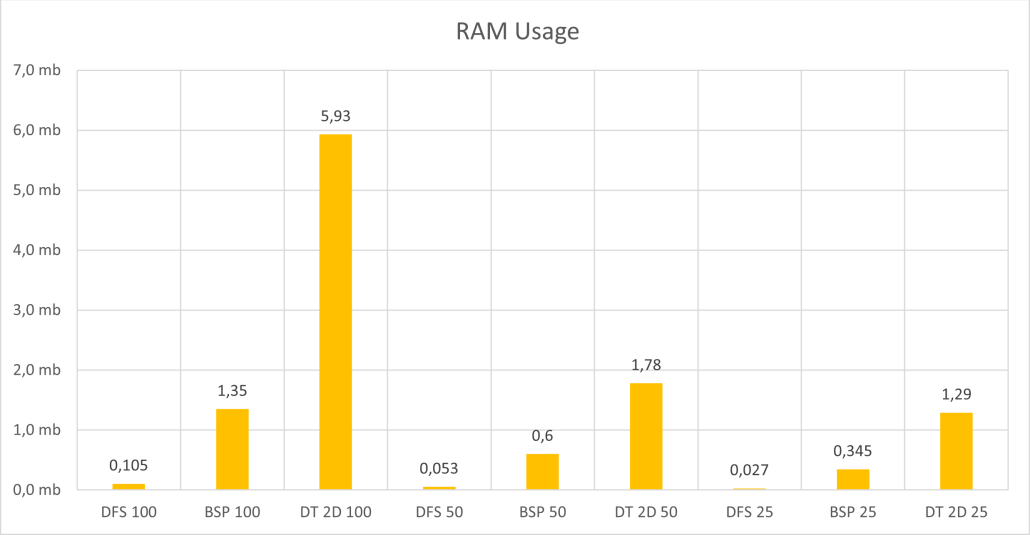

Figure 5.20: Zoomed in bar chart showing an overview of RAM usage for all algorithms using W11 hardware.

5.2.3.2 W10

As seen in the Figure, the RAM usage between different dungeon sizes changes relatively proportionally, except for DT 3D. As the RAM usage increases by over 4000% when going from 50 to 100 rooms. Between 25 to 50 rooms, it increases by about 18 per cent. When going from 100 to 50 rooms, the RAM usage decreased by about 50 per cent for DFS, BSP, and DT 2D, and going from 50 to 25 rooms RAM usage decreased by about 50 per cent again. For more accurate data, see the Table 5.8 above.

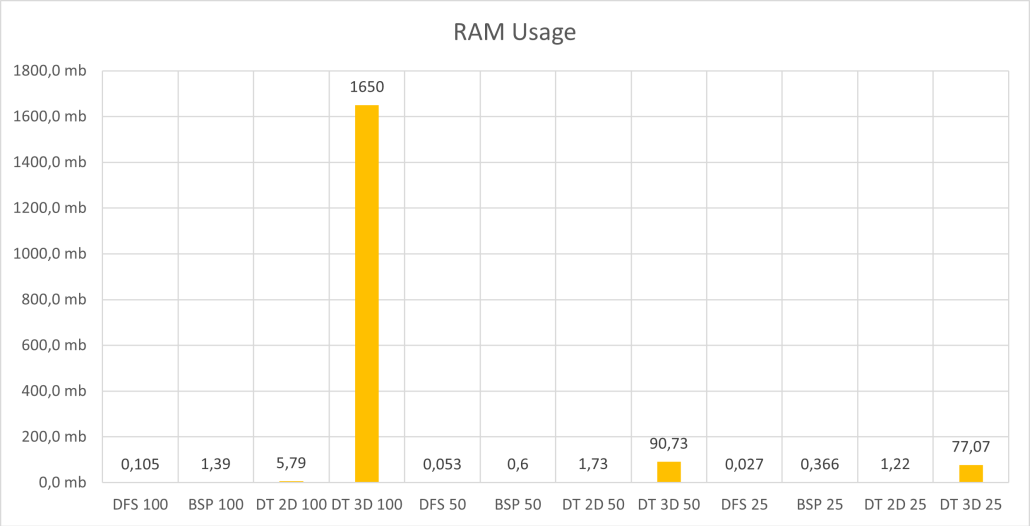

Figure 5.21: Bar chart showing an overview of RAM usage for all algorithms using W10 hardware.

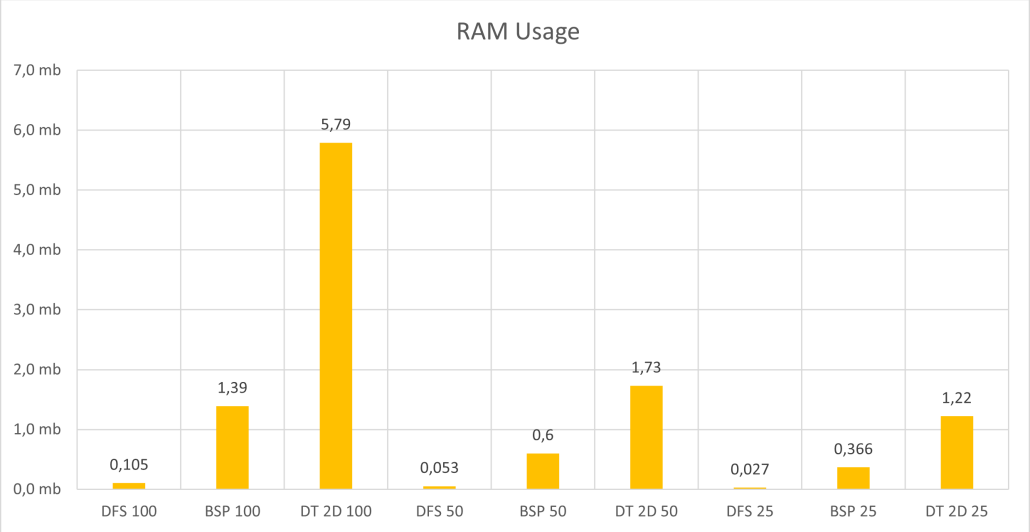

Figure 5.22: Zoomed in bar chart showing an overview of RAM usage for all algorithms using W10 hardware.

As stated earlier, in both the CPU usage and execution time sections, the results are mostly in line with what was concluded in those sections. DT 3D should not use such a high amount of RAM when generating 100 rooms. The possible causes have already been discussed in previous sections

5.3 Conclusion

Conclusions could be drawn based on the three research questions that this work was set to answer: find performance differences between BSP, DFS, and DT algorithms when used for procedural dungeon generation, the advantages and disadvantages of each algorithm, and how hardware affects performance.

When it comes to overall performance, the DFS algorithm was the superior choice across all tested dungeon sizes, excelling in all categories. DFS showcased exceptional efficiency, utilising less than half of the CPU and RAM compared to the rest of the algorithms. Additionally, it was also twice as fast as the next most efficient algorithm, DT 2D. For scenarios where maximum performance is necessary, DFS is the optimal choice, as its performance scales proportionally with the dungeon size. DT 2D, while not as efficient as DFS, ranked second overall in most categories for all tested dungeon sizes. It had the second-lowest CPU usage and was the second-fastest algorithm, but it consumed the third most RAM. In comparison to DFS, DT 2D exhibited over double the CPU usage, consumed more than 30 times the amount of RAM, and was 2.3 times slower than DFS. When generating 100 rooms, DT 2D's consumed more than three times the resources of DFS. BSP ranked third in performance across all tested dungeon sizes. While BSP had the third-highest CPU usage and was the third-fastest algorithm, it exhibited the second-lowest RAM usage. Compared to DFS, BSP's CPU usage was 7.5 times higher, its RAM usage was more than 65 times greater, and it performed 15 times slower. Lastly, DT 3D was the least efficient algorithm in all categories across all dungeon sizes, particularly when

generating 100 rooms. It was the most CPU-intensive algorithm, had the highest RAM usage, and was by far the slowest algorithm. When compared to DFS, DT 3D showcased 15 times higher CPU usage, consumed over 15,500 times more RAM, and performed 5,300 times slower.

Each algorithm had some advantages and disadvantages. DFS has extremely low-performance requirements. It uses less than 10% of the CPU when generating 100 rooms, takes 24 MS to generate a dungeon, and uses less than one MB of RAM. It is simple to implement and easy to understand. Some drawbacks are that the generated dungeon does not have any corridors, and it may be slightly monotonous, considering that all rooms look nearly identical and are the same size. This could be changed by using different prefabs to make the rooms look different, but it is something that does not exist by default and needs to be added.

DT 2D has low-performance requirements. It uses about 20% of the CPU when generating 100 rooms, takes 100 milliseconds to generate a dungeon, and uses six MB of RAM. The rooms in the dungeon can be of different sizes and have longer, more interesting corridors. Some drawbacks it has are that it is harder to implement and harder to understand and that the algorithm does not have any code that places game assets, such as prefabs, in the generated dungeon; as such, one needs to write extra code to add them to make it look presentable.

BSP has, for the most part, low-performance requirements. It has a medium-high CPU usage at nearly 50% when generating 100 rooms, takes 369 MS to generate a dungeon, and uses 1.35 MB RAM. The rooms in the dungeon can be of different sizes. It is relatively simple to implement and understand. Some drawbacks it has are that the corridors generated are straight, simple, and uninteresting. The algorithm does not have any code that places game assets, such as prefabs, in the generated dungeon; as such, one needs to write extra code to add them to make it look presentable. The current algorithm is not as optimised as it could be and needs to be improved.

DT 3D performance varies by room size. The performance required when generating less than 100 rooms is acceptable, but it uses up to 90% of the CPU, takes a few seconds to generate, and uses up to 90 MB of RAM. At 100 rooms, it is completely unusable as it maxes out the CPU, takes minutes to generate the dungeon, and uses GB of RAM. The dungeon generated is more interesting as it adds elevation to the dungeon, as it uses height instead of being completely flat, and generates more interesting corridors. Some drawbacks it has are that it is harder to implement and harder to understand. The algorithm does not have any code that places game assets, such as prefabs, in the generated dungeon; as such, one needs to write extra code to add them to make it look presentable.

Except for execution time, the newer and older computer hardware exhibited no substantial impact on the results. The new hardware generated dungeons at twice the speed of the older Hardware. CPU usage and RAM usage were within a few per cent of each other. This is to be expected as the biggest difference in hardware was speed-related. When it comes to the CPU, the Ryzen 5 has a 25% higher clock speed than the older I-5; it also has two fewer cores and eight fewer threads, and as such, it should be faster in every case. When it comes to RAM, the newer DDR4 has twice the clock speed and a higher bandwidth than the older DDR3 but has a higher latency.

Chapter 6

Discussion

6.1 Summary

This study aimed to compare the performance, advantages, and disadvantages of four different procedural dungeon generation algorithms: DFS, BSP, DT 2D, and DT 3D. These algorithms were tested against multiple different metrics, including CPU usage, execution time, and RAM usage. The result from the test demonstrated that DFS outperforms all other algorithms in terms of efficiency, requiring significantly less CPU and RAM while generating dungeons significantly faster. DT 2D is also an efficient algorithm, but it consumes more resources and generates the dungeon slower than DFS. In contrast, BSP and DT 3D showcased substantial drawbacks in terms of CPU and RAM usage, particularly for dungeons of larger sizes, with DT 3D being the least efficient of all the algorithms.

The hardware differences between W10 and W11 demonstrated a noticeable impact on execution time, with the newer W11 generating dungeons at twice the speed compared to the older W10, though CPU and RAM usage showed minimal differences between the two systems.

6.2 Performance

The performance differences between the algorithms can largely be attributed to their respective complexities. DFS, with its O(v + e) time complexity, is the most efficient algorithm, requiring fewer resources to generate dungeons of various sizes. This is evident in its remarkably low CPU and RAM usage. DFS's simplicity and low computational overhead make it a valid choice for scenarios where efficiency is of importance, particularly for smaller dungeons.

DT 2D, with its O(n log n) time complexity, is slightly more resource-intensive, likely due to its need to generate corridors and potentially more complex room layouts. Despite being less efficient than DFS, DT 2D offers more interesting, visually appealing dungeon structures due to its ability to generate complex corridors and rooms. This makes it preferable in applications where aesthetics matter, but resources are still a concern.

BSP's higher CPU usage could be related to how recursive space partitioning works, especially if the algorithm's implementation is not as optimised as it could be. DT 3D with its O(n2) time complexity seems to be the most computationally demanding, which could be attributed to the added complexity of working in a 3D space and generating both corridors and stairs. As noted, DT 3D performs well with smaller dungeon sizes, but its efficiency drastically declines when scaling up to larger dungeons, making it unsuitable for high-performance scenarios or large-scale dungeon generation. The high RAM usage and slow execution time for DT 3D when creating dungeons with 100 rooms are likely due to inefficiencies in the algorithm's implementation or inherent scaling problems with its approach.

6.3 Limitations

There are several limitations to this study. The performance was measured using a relatively small set of dungeon sizes (25, 50, 100 rooms). Testing with larger dungeon sizes might yield different results, particularly for DT 3D, which demonstrated performance degradation at higher room counters. Additionally, different settings used for dungeon generations may also drastically affect the performance with the same dungeon size. While hardware differences were explored, only 2 systems were used. Future studies should expand the hardware configuration to include more variations, such as different CPU and RAM configurations, to better understand how diverse systems affect performance.

Another limitation is the implementation of the algorithms themselves. Differences in algorithm optimisation could account for some of the performance disparities observed, especially with BSP. A more refined, optimised implementation of BSP may reduce its CPU usage and execution time.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

This study aimed to compare the performance, advantages and disadvantages of four different procedural dungeon generation algorithms DFS, BSP, DT 2D and DT 3D. The tested performance metrics were CPU usage, execution time and RAM usage using two different hardware configurations (W10 and W11).

The results demonstrated that DFS was the most efficient algorithm, overall, requiring significantly less CPU and RAM while generating dungeons significantly faster. This makes it a valid choice for scenarios demanding high performance and minimal resource consumption, particularly for smaller dungeons. DT 2D, while slightly less efficient, offered a balance between performance and aesthetically complex dungeon structures with varied room sizes and corridors. BSP, despite its higher CPU usage, remained relatively efficient when it comes to RAM usage but showed notable performance issues due to a less-than-optimal implementation. Finally, DT 3D, with its high complexity, proved to be the least efficient algorithm, particularly when generating larger dungeons, and showed severe performance degradation due to high CPU and RAM usage.

While the hardware configurations showed some difference in execution time, with W11 being approximately two times faster than W10, CPU and RAM usage between the two systems were largely the same. This indicated that the primary performance differences are related to the algorithm itself rather than related to hardware improvements, aside from faster execution time with newer hardware.

Ultimately, DFS emerged as the most robust algorithm for procedural dungeon generation in terms of efficiency, while DT 3D was found to be more suitable for smaller-scale or aesthetically driven dungeons.

7.2 Future Work

This study provides a comparison of four procedural dungeon generation algorithms, but there are several different things that can be further researched.

More extensive testing on how different dungeon generation settings impacted the algorithms' performance. This could give insight into how to optimise the algorithm, making it more efficient.

Further research into the optimisation of the algorithms used may also be needed, as the implementation used in the study is not optimised and could be improved. This is especially true for BSP and DT 3D, as they need more refined implementations to reduce CPU and RAM usage.

References

[1] "Basic bsp dungeon generation," https://www.roguebasin.com/index.php/Basic_BSP_Dungeon_generation, August 2023, accessed: 2025-06-06.

[2] P. Dinh, "Procedural dungeon generation algorithm explained," Phigames, 2013, accessed: 2025-03-25. [Online]. Available: https://www.reddit.com/r/gamedev/comments/1dlwc4/procedural_dungeon_generation_algorithm_explained/

[3] P. V. M. Dutra, S. M. Villela, and R. Fonseca Neto, "A mixed-initiative design framework for procedural content generation using reinforcement learning," *Entertainment Computing*, vol. 52, p. 100759, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1875952124001277

[4] H. Games, "No man's sky," 2016, video game, PC.

[5] G. Glorian, A. Debesson, S. Yvon-Paliot, and L. Simon, "The Dungeon Variations Problem Using Constraint Programming," in *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), L. D. Michel, Ed., vol. 210. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 27:1–27:16. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2021.27

[6] N. M. Husnul Habib Yahya, H. Fabroyir, D. Herumurti, I. Kuswardayan, and S. Arifiani, "Dungeon's room generation using cellular automata and poisson disk sampling in roguelike game," in *2021 13th International Conference on Information & Communication Technology and System (ICTS)*, 2021, pp. 29–34.

[7] O. Karlsson, "Procedurellt genererade dungeonkartor för roguelikespel : En jämförelse mellan binary space partitioning och delaunay triangulation," Master's thesis, University of Skövde, School of Informatics, 2019.

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

[9] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[10] ——, "Noise hardware," *Real-Time Shading SIGGRAPH Course Notes*, vol. 8, p. 19, 2001.

[11] Peter, "Procedural dungeon in unity 3d tutorial," Sunny Valley Studio,

2019, youTube video. [Online]. Available: https://www.youtube.com/watch?v=VnqN0v95jtU

[12] P. A. Putra, J. T. Tarigan, and E. M. Zamzami, "Procedural 2d dungeon generation using binary space partition algorithm and l-systems," in *2023 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, 2023, pp. 365–369.

[13] N. Shaker, A. Liapis, J. Togelius, R. Lopes, and R. Bidarra, "Constructive generation methods for dungeons and levels," in *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2016, pp. 31–55.

[14] SilverlyBee, "Procedural dungeon generator in unity [tutorial]," 2021, youTube video. [Online]. Available: https://www.youtube.com/watch?v=gHU5RQWbmWE

[15] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, 2014.

[16] Vazgriz, "Procedurally generated dungeons," Double Flash Studios, 2019, accessed: 2025-03-25. [Online]. Available: https://vazgriz.com/119/procedurally-generated-dungeons/

[17] ——, "Procedurally generated 3d dungeons," Double Flash Studios, 2021, youTube video. [Online]. Available: https://www.youtube.com/watch?v=rBY2Dzej03A

[18] M. Werneck and E. 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.

Appendix A

Newer Hardware

A.1 Summary Box Diagram

Algorithm	CPU (%)	Execution Time (s)	RAM (MB)
DFS 100	**6.500**	**0.024**	**0.105**
BSP 100	48.610	0.369	1.350
DT 2D 100	20.480	0.100	5.930
DT 3D 100	99.450	128.000	1635.000
DFS 50	**3.580**	**0.013**	**0.053**
BSP 50	21.620	0.098	0.600
DT 2D 50	7.440	0.031	1.780
DT 3D 50	87.760	3.018	91.100
DFS 25	**2.170**	**0.008**	**0.027**
BSP 25	14.520	0.060	0.345
DT 2D 25	6.420	0.026	1.290
DT 3D 25	83.790	2.175	81.610

Table A.1: Comparison of different algorithms and their effects on CPU, Execution Time, and RAM usage rounded to three decimals.

A.2 Summary Per Room

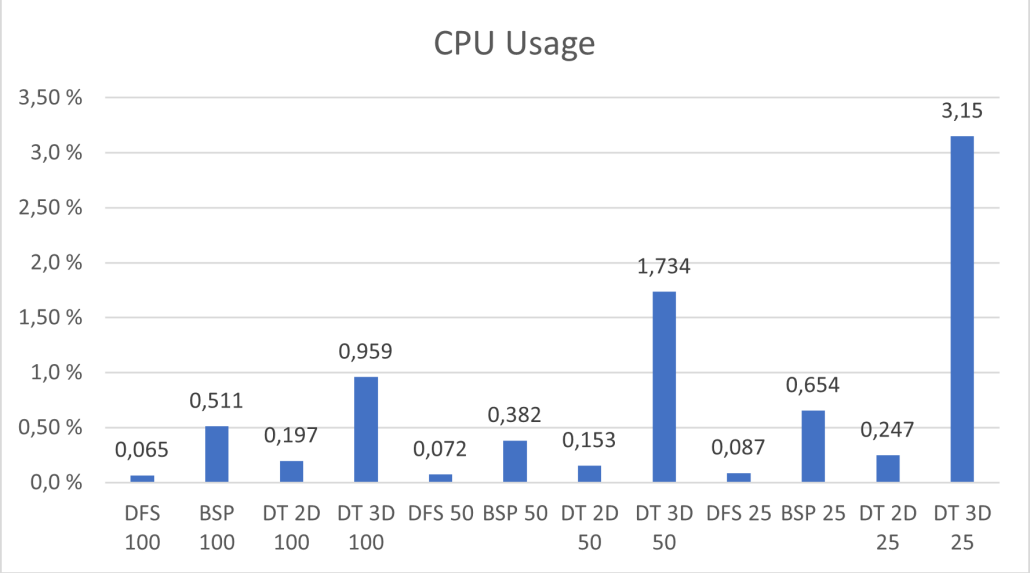

Figure A.1: Bar chart showing an overview of W11 CPU usage per room for all algorithms.

Algorithm	CPU (%)	Execution Time (ms)	RAM (kb)
DFS 100	**0.065**	**0.245**	**1.053**
BSP 100	0.511	3.872	14.181
DT 2D 100	0.197	0.958	57.074
DT 3D 100	0.959	1234.332	15767.000
DFS 50	**0.072**	**0.260**	**1.067**
BSP 50	0.382	1.734	10.601
DT 2D 50	0.153	0.638	36.626
DT 3D 50	1.734	59.650	1800.000
DFS 25	**0.087**	**0.311**	**1.096**
BSP 25	0.654	2.710	15.524
DT 2D 25	0.247	0.998	49.615
DT 3D 25	3.150	81.764	3068.000

Table A.2: Comparison of different algorithms and their effects on CPU, Execution Time, and RAM usage per room rounded to three decimals.

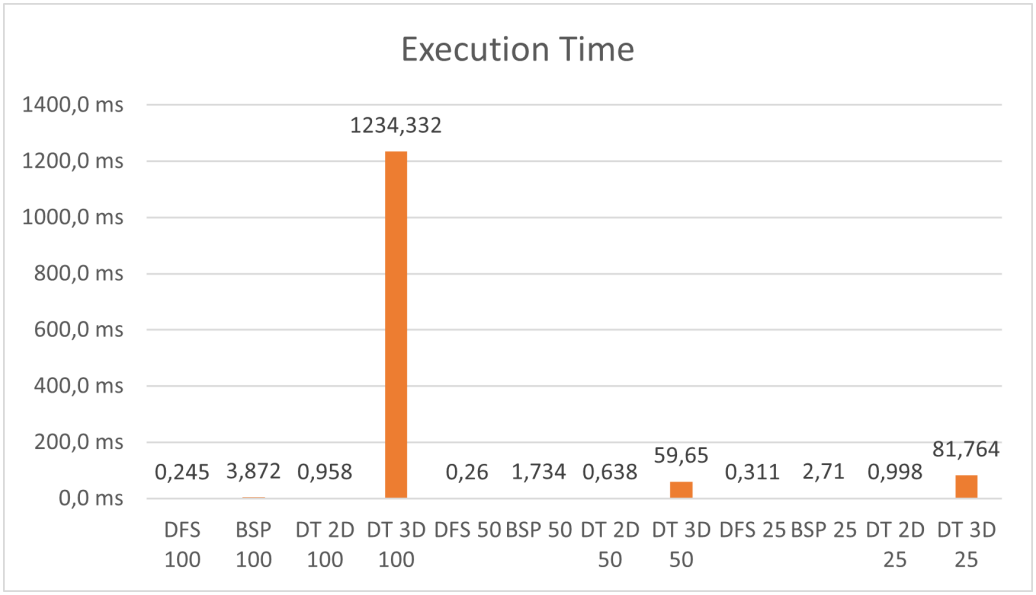

Figure A.2: Bar chart showing an overview of W11 execution time per room for all algorithms.

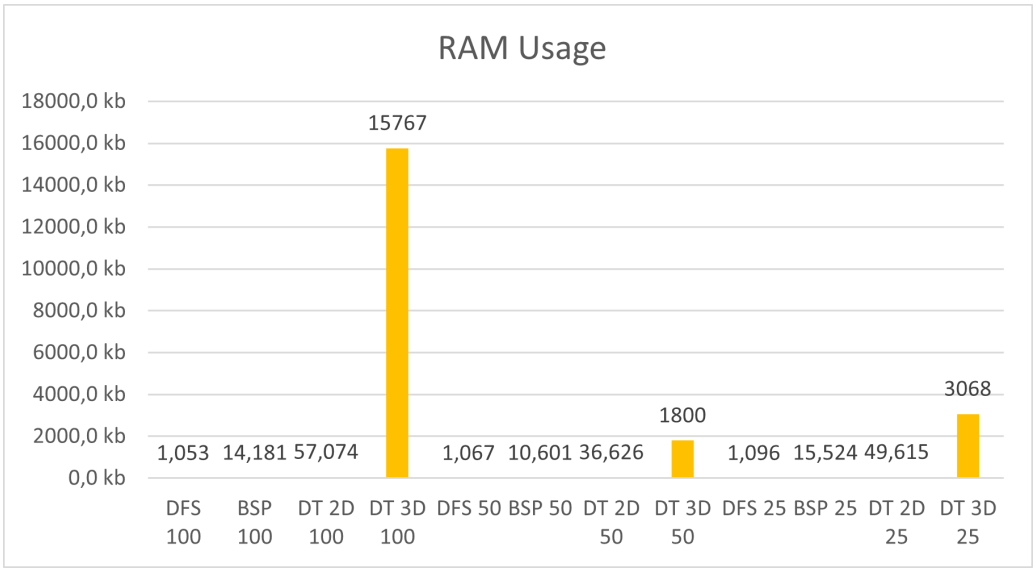

Figure A.3: Bar chart showing an overview of W11 RAM usage per room for all algorithms.

Appendix B

Older Hardware

B.1 Summary Box Diagram

Algorithm	CPU (%)	Execution Time (s)	RAM (MB)
DFS 100	**6.650**	**0.039**	**0.105**
BSP 100	45.890	0.564	1.390
DT 2D 100	19.760	0.162	5.790
DT 3D 100	99.60	222.772	1650.000
DFS 50	**3.830**	**0.021**	**0.053**
BSP 50	20.460	0.160	0.600
DT 2D 50	7.260	0.051	1.730
DT 3D 50	88.440	5.111	90.730
DFS 25	**2.250**	**0.012**	**0.027**
BSP 25	15.060	0.108	0.366
DT 2D 25	6.310	0.042	1.220
DT 3D 25	81.800	3.009	77.070

Table B.1: Comparison of different algorithms and their effects on CPU, Execution Time, and RAM usage.

B.2 Summary Per Room

Algorithm	CPU (%)	Execution Time (ms)	RAM (kb)
DFS 100	**0.067**	**0.390**	**1.051**
BSP 100	0.496	6.086	15.011
DT 2D 100	0.185	1.520	54.264
DT 3D 100	0.942	2107.587	15610.000
DFS 50	**0.077**	**0.417**	**1.065**
BSP 50	0.382	2.994	11.194
DT 2D 50	0.145	1.028	34.669
DT 3D 50	1.801	104.095	1848.000
DFS 25	**0.090**	**0.490**	**1.096**
BSP 25	0.635	4.548	15.453
DT 2D 25	0.244	1.630	47.104
DT 3D 25	2.953	108.642	2782.000

Table B.2: Comparison of different algorithms and their effects on CPU, Execution time, and RAM usage per room rounded to three decimals.

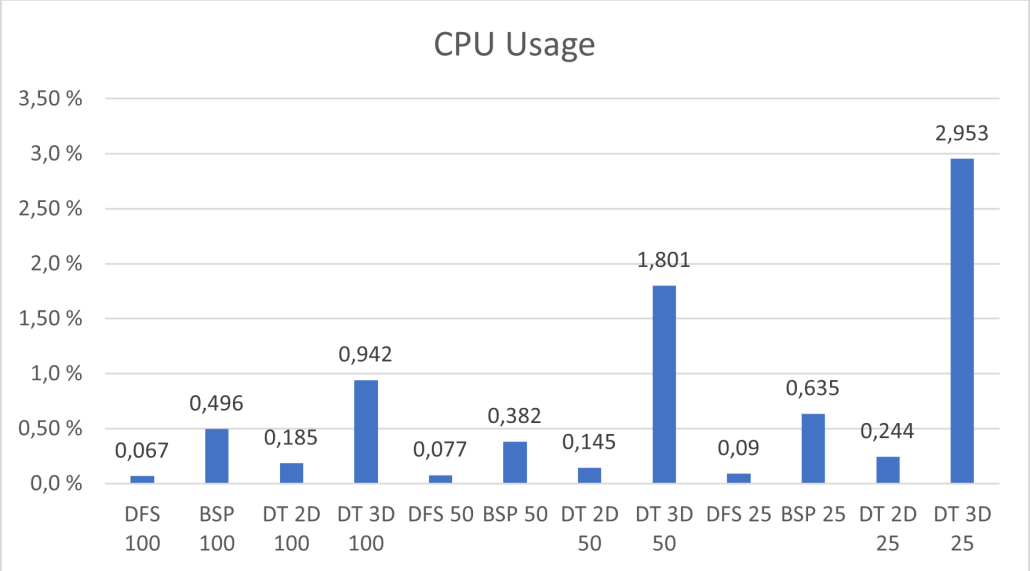

Figure B.1: Bar chart showing an overview of W10 CPU usage per room for all algorithms.

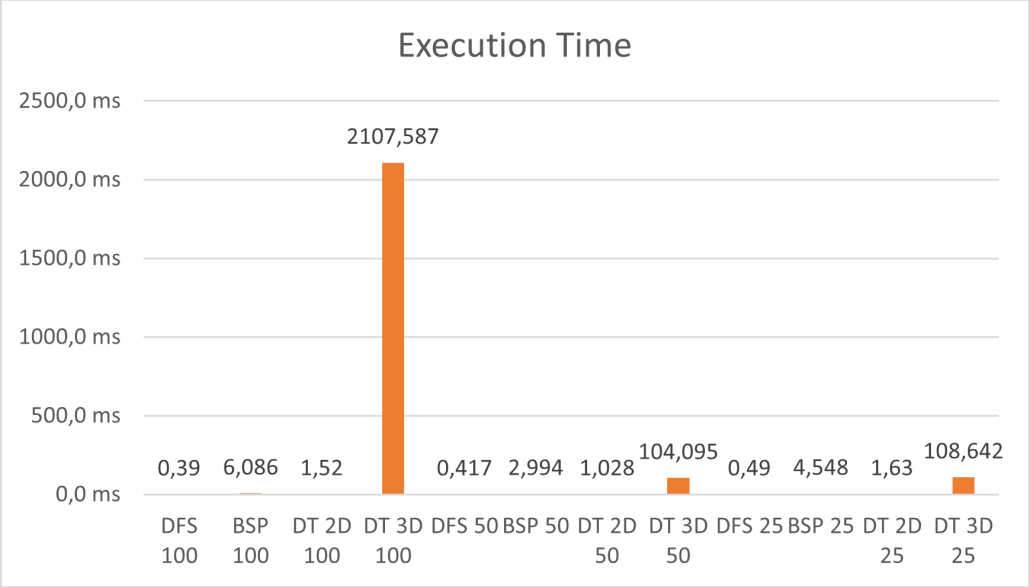

Figure B.2: Bar chart showing an overview of W10 execution time per room for all algorithms.

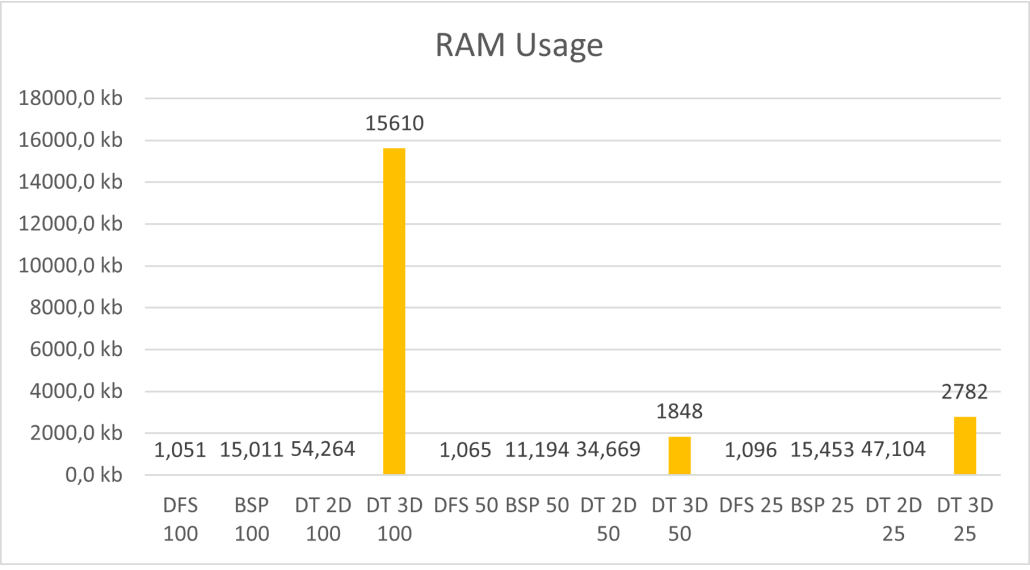

Figure B.3: Bar chart showing an overview of W10 RAM usage per room for all algorithms.

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden