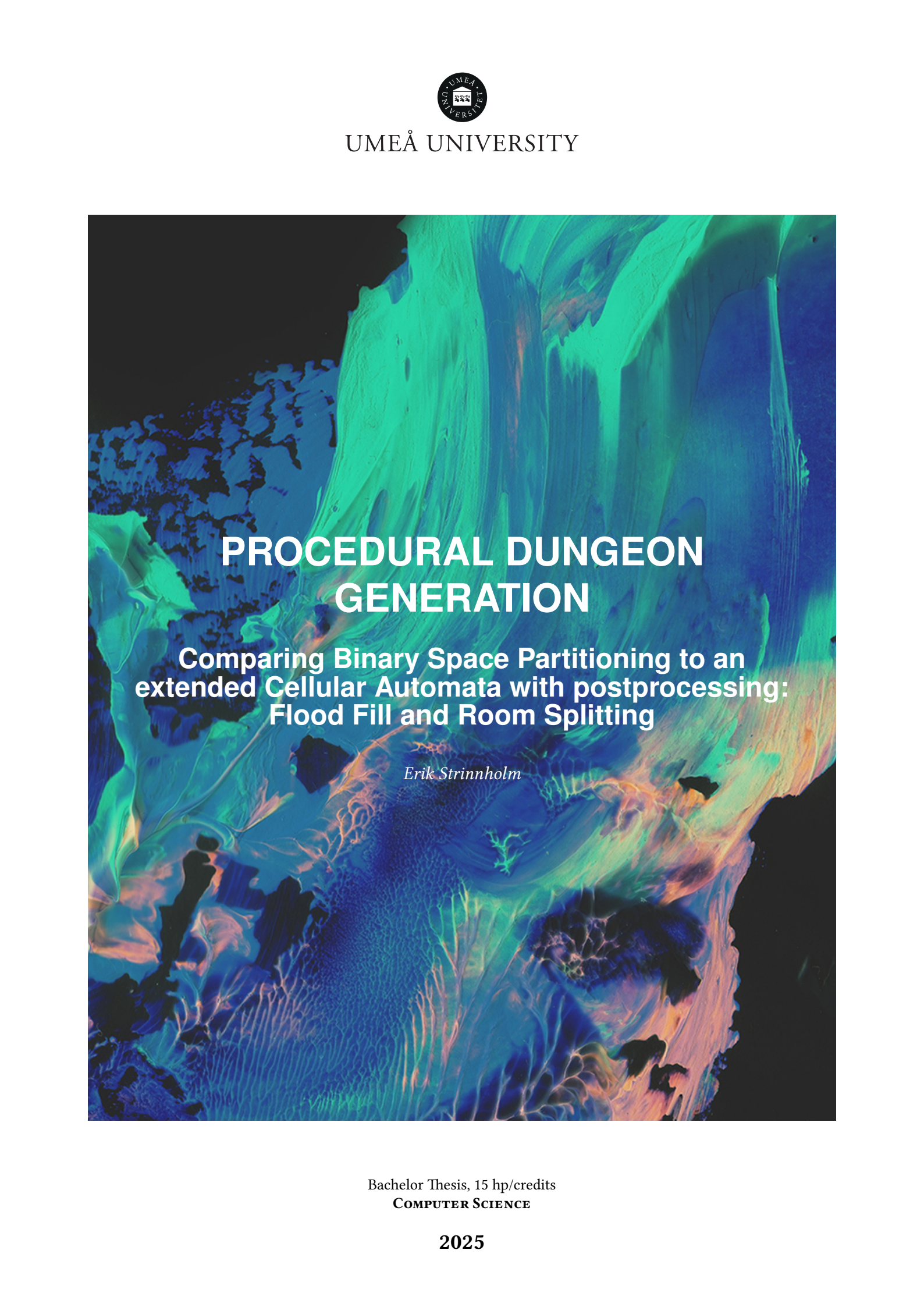


UMEÅ UNIVERSITY

PROCEDURAL DUNGEON GENERATION

Comparing Binary Space Partitioning to an extended Cellular Automata with postprocessing: Flood Fill and Room Splitting

Erik Strinnholm

Bachelor Thesis, 15 hp/credits

Computer Science

2025

Abstract

This thesis compares two Procedural Dungeon Generation (PDG) approaches building on Binary Space Partitioning (BSP) and Cellular Automata (CA) regarding three PDG properties: expressivity, reliability and controllability. To enable a fair comparison, the CA method is extended with two post-processing steps: a flood fill algorithm to detect rooms, and a room splitter to enforce room size constraints. Both methods are then used to generate 2D top-down dungeon layouts within a 50x50 tile map.

The evaluation uses two agent-based metrics: Difficulty, how much the agent must explore to find the best path from the start room to the end room. Diversity, how different the agent's actions are when solving different layouts. The agent uses the A* pathfinding algorithm to navigate the dungeon layouts. The dungeon layouts are represented as graphs, where room are nodes and corridors are edges. A Minimum Spanning Tree (MST) is constructed to ensure room connectivity, and additional corridors are added selectively to increase layout complexity. Each configuration generates 1000 dungeons, and the results are visualized using expressive range analysis via 2D heatmaps of Difficulty and Diversity.

The results show that BSP is more reliable and predictable, consistently generating well-structured dungeons. In contrast, CA exhibits greater expressivity, producing a broader and more varied set of layouts, but at the cost of occasional reliability issues, such as isolated rooms or visual artifacts, the causes of which remain unresolved.

The choices between BSP and CA depends on design priorities: BSP is preferable for structured and predictable layouts, while CA might be better for scenarios demanding variety and emergent structure.

Contents

1 Introduction

Procedural Content Generation (PCG) has been used in games for decades and remains a popular technique today. Its roots can be traced back to tabletop games like Dungeon and Dragons (1974-present) [3], and it has been widely adopted in game development, particularly in genres such as roguelikes and role-playing games (RPGs) [17]. Notable examples include Rogue (1980), the Diablo series (1996-2023), and Minecraft (2009).

A specific branch of PCG is Procedural Dungeon Generation (PDG), which focuses on the algorithmic creation of dungeon layouts. In game development, the term "dungeon" typically describes a maze-like environment composed of rooms connected by corridors [16]. Given PDG's long history, many approaches for generating dungeons have been developed [17][13][9]. However, due to the subjectivity of player experience, different approaches may suit different design goals [13].

An ongoing discussion in PDG research, concerns what makes a "good" dungeon generator [13]. Evaluating dungeon generation algorithms can be challenging because there is no unified evaluation framework, which makes comparing different approaches difficult [9][2]. While some studies have proposed common evaluation metrics like Linearity (the degree to which the dungeon layout follows a linear path) and Leniency (the quantity of features, such as enemies or gaps etc, contributing to the game's difficulty compared to safe areas), these metrics are often calculated differently or tailored to specific game genres, limiting their general applicability [13][14].

Other work advocate for more general, agent based metrics like Difficulty and Diversity, which reflect player behaviors and are less domain-specific [2]. This thesis uses simulated agents to navigate the generated dungeons, from a start Room to a predefined End Room. The agent utilizes an A* pathfinding algorithm to explore the layout. By measuring much the agent must explore to find the best path to the goal (Difficulty) and how different the agent's actions are when solving different layouts (Diversity), the dungeon layouts can be evaluated from a player-centric perspective.

This thesis compares two approaches for generating 2D top-down dungeon layouts, based on two commonly used constructive PDG methods: Cellular Automata (CA) and Binary Space Partitioning (BSP) [13]. CA was extended in this work with additional post-processing steps, Room Detection (Flood Fill) and a Room-Splitter, to adress some of its limitations in controlling room size and a lack of structural rooms. The extension was made to make CA outputs more structurally comparable to BSP-generated layouts, thereby enabling a fair evaluation.

The evaluation focuses on three core properties of a dungeon generator: Reliability (it should produce valid dungeons consistently), Expressivity (it should generate a wide variety of distinct layouts), and Controllability (it should allow designers to influence outputs through parameters) [13]. To assess these properties, the thesis uses the two agent-based evaluation metrics Difficulty and Diversity as proposed by [2].

Therefore, this thesis systematically investigates how Cellular Automata and Binary Space Partitioning compare in terms of reliability, expressivity, and controllability when generating 2D top-down dungeon layouts, using difficulty and diversity as evaluation metrics.

2 Related Work

2.1 Overview

Several approaches for Procedural Dungeon Generation (PDG) exist and the paper [9] highlight the growing number of papers on the topics in recent years. The survey [17], identified two primary categories of PDG methods: Constructive Methods and Search-Based Methods.

Constructive methods generate dungeon layouts through direct, step-by-step rules. Examples of constructive approaches include Random Walk algorithms, Binary Space Partitioning, Cellular Automata and Generative Grammar. In contrast, Search-based methods build the dungeon as an optimization problem and search for layouts that maximize certain design goals. With the most notable example being Genetic Algorithms. Beyond these two categories, there are a number of hybrid approaches, combining and utilizing aspects between algorithms [17][9]. More recently there have also been a number of works on Machine Learning and Deep Learning techniques applied for PDG [10], although these approaches are not the focus of this thesis.

2.2 Challenges in PDG and Evaluation

Despite advancement in PCG techniques, PDG remains a complex problem, particularly when evaluating generated dungeons [13]. A good dungeon generator is expected to satisfy several properties such as: Reliability, it should produce valid dungeons consistently. Expressivity, it should be able to generate a wide variety of layouts. Controllability, it should allow designers to influence outputs through parameters [13].

One major challenge in PDG research is the lack of unified evaluation metrics. While certain metrics such as Linearity and Leniency are commonly used, they are often defined and calculated differently across studies, making comparison difficult [2].

The authors in [9] highlight the importance of considering a range of metrics, distinguishing between spatial metrics (such as map connectivity and layout structure) and player-centrics metrics (such as perceived difficulty). Measuring both could be important for a fuller understanding of a dungeon generator's performance.

One widely used framework for evaluating PDG is Expressive Range Analysis (ERA) [9]. ERA focuses on analyzing the variety of outputs a generator can produce as a whole, rather than evaluating individual layouts, providing a broader view of the generator's behaviour. Another commonly used tool is the use of heatmaps [9], which visualize how a generator's output changes in response to different input parameters, offering insight into a generator's controllability [13].

When selecting evaluation metrics, the authors of [13] recommends focusing on emergent properties, outcomes that arise indirectly from input parameters, rather than only on properties closely tied to the generator's immediate settings. This approach can lead to a deeper and more meaningful evaluation.

The authors of [2] proposes two agent-based metrics for evaluating procedural levels.

Diversity examines how different the agent's actions are when solving different levels, based on how much the action paths change. Difficulty is based on how much the agent has to search around before finding the best solution, harder levels make the agent explore more.

To organize and compare PDG approaches, the authors of [17] proposes a classification system, based on a refinement of the PCG taxonomy defined in [15]. Their categories include content need, generation time, generation control, generality, random choice, generation method, and content authorship. This taxonomy will be used later in this thesis to describe and compare the two methods (see Section 4.7).

2.3 Similar Works

The paper [14] explores the use of Answer-Set Programming (ASP) to create a mission graph for dungeon generation, they use a domain specific definition of mission-linearity, map-linearity, leniency and path redundancy. To evaluate their approach the authors use ERA and heatmaps.

The author of [1] studies six different combinations of room- and corridor-based constructive PDG algorithms, in a two step approach, although with less emphasis on the evaluation aspect of the approaches. The methods used include Random Room Placement and BSP Room Placement, with Random Point Connect, Drunkard's Walk, and BSP Corridors.

The work in [4], focuses on constructive PDG approaches, particularly dungeon furnishing, and uses ERA to evaluate more spatial centric properties like empty tile percentage, path length and wall count.

In summary, while there are many techniques for generating dungeons, evaluating them consistently and meaningfully remains a difficult and important part of PDG research. The aim of this thesis is to evaluate two constructive approaches, using ERA and heatmaps, with the metrics Diversity and Difficulty proposed by the authors of [2], as to give both a spatial and player-centric view of the generation process [9].

3 Background

3.1 PDG Taxonomy

To better understand and categorize different PDG techniques, the authors of [17] proposes a taxonomy that refines the work of [15]. The classification includes the following categories:

- Content Need: distinguishes between generation that is essential for gameplay, such as required levels, and optional generation, such as decorative items or bonus content.
- Generation Time: separates approaches that create content offline, prior to gameplay, from those that generate content online, during play.
- Generation Control: examines the degree to which the process can be directed, either through random seeds offering limited influence or through multiple adjustable parameters that allow more precise shaping of the results.
- Generality: refers to whether the content is designed to suit a general audience or adapted for individual players.
- Random Choice: distinguishes between deterministic methods that always produce the same output for given inputs and stochastic methods that introduce variation.
- Generation Method: classifies approaches as either constructive, building the content in a single pass, or generate-and-test, where outputs are repeatedly tested and refined.
- Content Authorship: considers the level of human involvement, ranging from fully automatic generation to mixed-initiative approaches where designers directly influence aspects of the content beyond basic parameter tuning.

3.2 Constructive Approaches

Constructive methods for PDG build content step-by-step through direct algorithms that iteratively build dungeon layouts. In this thesis two constructive methods were chosen due to their complementary nature. BSP tends to produce structured, clearly divided layouts suited for organized dungeon spaces [13], while CA generates more chaotic, cavern-like formations that resemble natural caves [16][7].

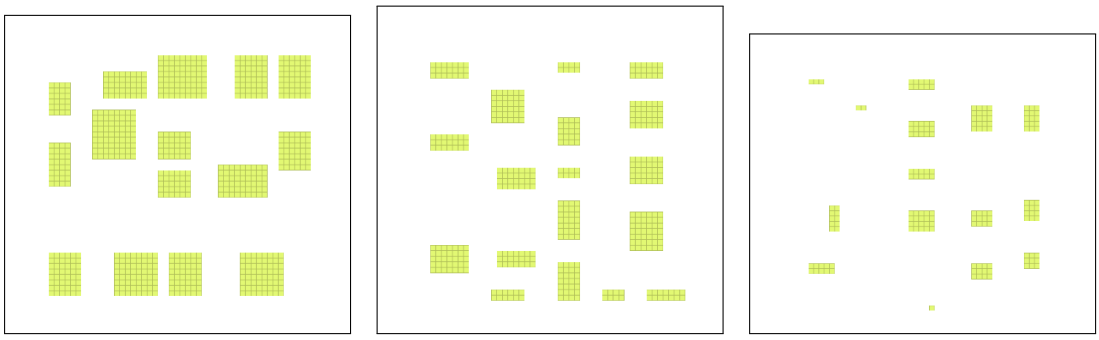
3.3 Binary Space Partitioning

Binary Space Partitioning (BSP) is a method originally developed for 3D computer graphics graphics [12], later adapted for procedural content generation (PCG) in games. In the context of Procedural Dungeon Generation (PDG), BSP is used to recursively divide a two-dimensional space into smaller, non-overlapping regions, creating an organized structure suitable for room layouts.

In this thesis, BSP is applied to segment the dungeon map into rectangular regions, with a constraint on the Minimum Width and Height allowed for any resulting subregion. The algorithm begins with the full dungeon space and iteratively splits it into two parts, either horizontally or vertically, at a randomly chosen point. The direction of each split is also selected randomly, introducing variability into the otherwise structured division. Each subregion is added to a queue, and the process continues until no region can be further divided without violating the minimum size constraints. In theory this sets the Maximum Width to (Minimum Width x 2 - 1) and the Maximum Height to (Minimum Height x 2 -1), but it depends on the random split if one gets rooms that big. The resulting structure can be represented by a binary tree, where the root node represents the original dungeon space and each internal node represents a split.

Unlike some variations of BSP that generate a room strictly equal to the size of the leaf region, this implementation may optionally carve out a smaller room within the bounds of each final region, allowing for an offset to prevent clipping between rooms. An offset of 1 means that each BSP room is shrunk by 1 tile on each side. (see Figure 1A,1B,1C for different offset configurations).

While some previous work, such as [13], use BSP not only for room division but also to directly build corridors between rooms connecting sibling nodes under the same parent, this thesis adopts a more general corridor construction approach. Instead of strictly linking sibling rooms, corridors are created later by generating a Minimum Spanning Tree across all room nodes (see Section 4.3), before constructing additional corridors. This should allow for more flexible and natural dungeon connectivity beyond the immediate parent-child structure imposed by BSP-trees. By introducing controlled randomness into the partitioning axis and split point, the idea is that the method becomes more varied and creates a more diverse and organic room arrangement while keeping the organized characteristics of BSP.



(A) (B) (C)

Figure 1: Shows the BSP algorithm generated rooms: A) Min Size=6x6 and Offset=1, B) Min Size=6x6 and Offset=2, C) Min Size=6x6 and Offset=3.

3.4 Cellular Automata

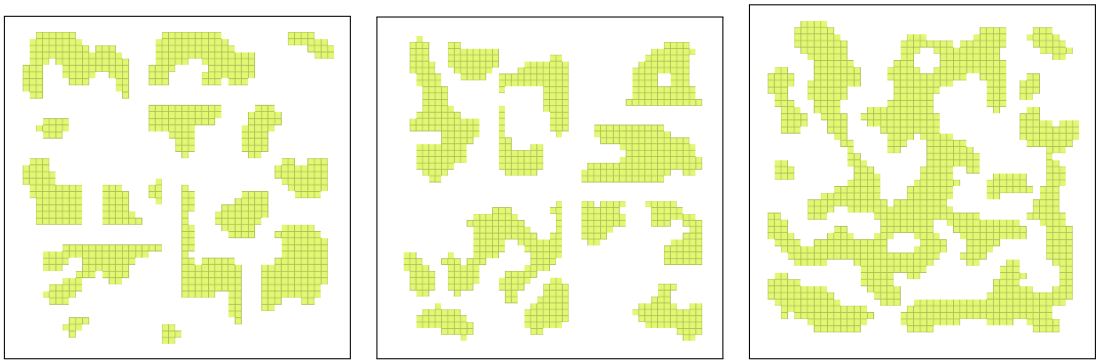
Cellular Automata (CA), initially proposed by [18], are computational models adapted across fields such as computer science, physics, and biology to simulate phenomena like growth, development, and physical processes. In the context of Procedural Dungeon Generation (PDG), CA is often used to produce sporadic, cavern-like environments [16].

This thesis builds on the CA-based approach described by [7], which was originally designed for real-time generation of infinite cave maps in games. In their method, a 2D grid is initialized with a random distribution of solid (rock) and walkable (floor) cells. Iteratively, the state of each cell is updated based on the number of solid neighbors using a simple rule: a cell becomes rock if the number of surrounding rock tiles (based on the Moore neighborhood) exceeds a threshold T; otherwise, it becomes floor. This process is repeated for a fixed number of iterations n, gradually shaping the layout into cave-like structures. The parameters used to control the generation include the initial rock density r, the threshold T, and the number of iterations n.

While this process is good at producing varied and naturalistic spaces, it lacks built-in constraints for controlling room size or shape [16][13]. CA generated spaces often contain large, unbounded spaces, which can lead to unbalanced spatial spaces, particularly when trying to identify playable room areas.

To address this, this thesis introduces Flood Fill and a Room Splitter as post-processing steps. Flood Fill is used to detect discrete room areas within the generated layout as rooms, while the room splitter is applied to split any oversized room regions. The divider inspects each identified room's shape and divides it along its longest axis, introducing a configurable spacing offset between sub-regions to prevent visual or spatial overlap. This division continues until all rooms meet a maximum tile count constraint.

This adjustment introduces greater controllability to the CA-based generation and allows a fairer comparison with the BSP approach, which naturally limits room sizes, allowing both to use similar input parameters for the generation process. However, a non-constraint CA is included in the generated dungeon configurations for comparison, to compare both approaches. See Figure 2(A-C) for a comparison between the configurations.



(A) (B) (C)

Figure 2: Shows the CA approach with the following parameters: Initial Fill Percentage=50%, Iterations=3, NeighborhoodShape=Moore(3x3) TransitionRule=4 and Offset=1. In addition to: A) RoomSplitter= max 100 cells (medium room), B) RoomSplitter = max 150 cells, (large room) and C) RoomSplitter= no limit (CA default).

3.4.1 Flood Fill

Flood Fill is an well established algorithm that systematically explores connected areas within a space, identifying bounded or continuous regions. In dungeon generation, it is particularly useful for detecting rooms, validating reachability, or analyzing overall map connectivity. In the context of CA-generated layouts, Flood Fill is used as a post-processing step to identify contiguous clusters of walkable tiles—cells marked as "empty"—and treat these clusters as rooms (see Algorithm 1).

The algorithm starts at an unvisited, walkable tile and uses a queue-based breadth-first search (BFS) to explore neighboring tiles. Only orthogonal directions (north, south, east, west) are considered, to ensure more clearly defined room boundaries. For each tile in the queue, its unvisited neighbors are checked. If they are also walkable, they are marked as visited and added to the queue. This process continues until the entire connected region has been visited. The group of tiles discovered forms a room, which is then added to the list of identified rooms.

Using Flood Fill for CA help extract rooms from the grid generated by the CA algorithm.

Algorithm 1: Flood Fill to detect Rooms

Input: 2D map of size $W \times H$, where 0 = walkable, 1 = wall
Output: List of rooms (each as a set of connected tiles)

```
Initialize visited[W][H] as false;
rooms ← empty list;
foreach cell (x, y) in the map do
    if map[x][y] = 0 and visited[x][y] = false then
        Initialize queue Q ← [(x, y)];
        Mark (x, y) as visited;
        roomTiles ← ∅;
        while Q is not empty do
            t ← Q.pop();
            Add t to roomTiles;
            foreach neighbor n of t (up, down, left, right) do
                if n is within bounds, map[n] = 0, and not visited then
                    Mark n as visited;
                    Enqueue n into Q;
        Add roomTiles as a new room to rooms list;
return rooms;
```

3.4.2 Room Splitting

To introduce controllability to the CA-generated layouts, this thesis uses a custom room splitting strategy as a post-processing step. This method, inspired by BSP-style subdivision (see Section 3.3), ensures that no individual room exceeds a specified tile count, which would otherwise lead to overly large or unbalanced spaces.

The algorithm works recursively. For each detected room, it first checks whether the total number of tiles exceeds the configured maximum. If it does, the algorithm calculates the room's bounding box and determines the longer axis—either width or height. The room is then split along that axis at its midpoint, applying a configurable spacing offset to ensure visual and spatial separation between the resulting subregions. These new subrooms are then

re-queued and checked in the same way. This process continues until all rooms are within the allowed size limit. See Algorithm 2 for psudocode of the algorithm.

This algorithm ensure all room from CA layouts fall within the defined maximum size, improving CA's controllability and making the CA-based generation more comparable to BSP.

Algorithm 2: Room Splitting

Input: List of rooms (sets of tiles), SizeLimit (tile count), Offset
Output: List of rooms all within the size limit

```
Initialize result list as empty;
Initialize queue Q with all input rooms;
while Q is not empty do
    room ← Q.pop();
    if room has more than SizeLimit tiles then
        Bounding box: minX, maxX, minY, maxY;
        width ← maxX − minX;
        height ← maxY − minY;
        if width ≥ height then
            splitX ← minX + width / 2;
            leftTiles ← tiles where x ≤ splitX − Offset;
            rightTiles ← tiles where x ≥ splitX + Offset;
            if leftTiles not empty then
                enqueue room from leftTiles into Q;
            if rightTiles not empty then
                enqueue room from rightTiles into Q;
        else
            splitY ← minY + height / 2;
            bottomTiles ← tiles where y ≤ splitY − Offset;
            topTiles ← tiles where y ≥ splitY + Offset;
            if bottomTiles not empty then
                enqueue room from bottomTiles into Q;
            if topTiles not empty then
                enqueue room from topTiles into Q;
    else
        Add room to result list;
return result list;
```

3.5 Graph Theory

This thesis uses an abstract representation of the generated dungeons, it models the generated rooms as nodes in an undirected graph $G = (V, E)$, where each node $v \in V$ corresponds to a room, and each edge $e \in E$ represents a potential corridor between two rooms. Using this representation allow the use of graph algorithms to build a Minimum Spanning Tree (MST) to connect the rooms with corridors. A MST is a subset of the edges that connect all vertices in the graph without cycles and with the minimum possible total edge weight.

To construct the MST, this thesis uses a variation of Prim's Algorithm (see Algorithm 3). Prim's Algorithm was originally developed by [6] and later republished by [11]. The

algorithm, starts with a single node and grows the tree by repeatedly adding the shortest edge that connects a node inside the tree to a node outside of it. This thesis uses a custom heuristic: the distance between the closest pair of outer tiles between two rooms. It reflects the actual cost of connecting rooms in a grid-based dungeon layout. The resulting MST ensures that all rooms are connected with a minimal total corridor length, preventing isolated rooms.

Once the MST is established, additional corridors can then be added to introduce sub-loops, to give players more meaningful choices in navigation by improving the layout complexity. In the evaluation (Section 4.6), this graph-based model also allow the use of simulated agents (via A* pathfinding) to analyze player-centric metrics like Difficulty and Diversity, based on the structure of the generated graph [2].

Algorithm 3: MST, using Prim's Algorithm with Custom Heuristic

Input: List of rooms
Output: Set of corridor connections forming the MST

```
T ← empty list of corridors;
S ← list with one randomly selected starting room;
U ← all remaining rooms;
while U is not empty do
    minDistance ← ∞;
    bestPair ← null;
    foreach roomA in S do
        foreach roomB in U do
            (tileA, tileB, d) ← ClosestOuterTiles(roomA, roomB);
            if d < minDistance then
                minDistance ← d;
                bestPair ← (roomA, roomB);
                bestTiles ← (tileA, tileB);
    Add corridor between bestPair to T;
    Move roomB from U to S;
return T;
```

3.6 Corridor Building

Once the Minimum Spanning Tree (MST) has been established, the logical room connections must be translated into spatial corridors on the dungeon grid. To achieve this, this thesis uses the A* pathfinding algorithm (see Algorithm 4) to compute the shortest valid path between each selected room pair, treating all rooms and previously placed corridors as obstacles.

A* was originally developed by [5] as an extension of Dijkstra's algorithm. It is a best-first search algorithm that combines the actual cost from the start node, $g(n)$, with a heuristic estimate to the goal, $h(n)$, resulting in a total score: $f(n) = g(n) + h(n)$. The algorithm expands nodes based on the lowest f-score. This thesis uses Manhattan Distance as the heuristic function.

To improve performance and avoid excessive computation, this thesis includes a time limit and a step cap. The start and end tiles are also temporarily excluded from the obstacle set to enable valid corridor connections. Using an obstacle set helps the algorithm to respect existing geometry and build around existing rooms and corridors.

Algorithm 4: A* Pathfinding for Corridor Generation

Input: Start tile, End tile, Obstacle tiles, Map bounds
Output: Tile path between Start and End

```
Initialize openSet (priority queue) with start tile;
gCost[start] ← 0;
fCost[start] ← heuristic(start, end);
cameFrom ← empty map;
while openSet not empty and time/step limits not exceeded do
    current ← node in openSet with lowest fCost;
    if current = end then
        return path reconstructed from cameFrom;
    foreach neighbor of current within bounds do
        if neighbor is not in obstacles and not already evaluated then
            tentativeG ← gCost[current] + 1;
            if neighbor not in gCost or tentativeG < gCost[neighbor] then
                Update cameFrom, gCost, fCost for neighbor;
                Add neighbor to openSet;
return empty path (no connection found);
```

3.7 Dungeon Evaluation

The quality of generated dungeons is assessed using both agent-based evaluation metrics and expressive range analysis, providing a comprehensive understanding of how the two generation methods (BSP and CA) perform under different input configurations.

Following the approach in [2], this thesis uses two agent-based metrics: Difficulty and Diversity. Rather than focusing solely on structural features of the dungeons, these metrics examines how a simulated agent behaves when navigating from the Start Room to the End Room.

Difficulty measures how hard it is for an agent to find the correct path from the Start Room to the End Room. It is based on the amount of exploration the agent must perform before reaching the goal. Specifically, it examines how much the agent's search expands beyond the optimal path. A dungeon with a very straightforward, linear path will have low difficulty, as the agent requires little to no exploration. Conversely, dungeons with multiple branching corridors, dead-ends, and loops cause the agent to explore a larger portion of the map before finding the correct path, indicating higher difficulty.

$$\text{Difficulty} = \frac{\text{ExploredTiles}}{\text{TotalReachableTiles}}$$

Diversity measures the variation in the agent's movement across different dungeon layouts. The agent's directional move sequences (up, down, left, right) are recorded as strings for each dungeon instance, capturing the exact navigation path taken from the Start Room to the End Room. These action sequences are then compared pairwise between dungeons to evaluate how varied the agent's navigation is.

To compute this, this thesis following the approach by [2], uses the *Levenshtein Distance*, developed by [8], which calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one sequence into another. A dungeon lay-

out's diversity score is defined as the average Levenshtein Distance between the agent's movement sequences across five generated samples.

A large Levenshtein Distance between action sequences suggests that the agent takes noticeably different routes through each dungeon, reflecting a greater variation in layout design. Conversely, a low Levenshtein Distance would suggest that the generated layouts are structurally similar, leading to repetitive agent behavior. By comparing the average Levenshtein distance for a layout against a sample of other layouts you can see its comparative Diversity score.

$$\text{Diversity}(s_i) = \frac{1}{5} \sum_{j=1}^{5} \text{Levenshtein}(s_i, s_j) \quad \text{for } i \neq j$$

4 Methodology

4.1 Overview

The methodology consists of three main parts. First, it outlines the design choices made during the development of the two Procedural Dungeon Generation (PDG) approaches. Second, it describes the two step generation process: generating rooms and then connecting them with corridors. Finally, it explains how various evaluation metrics are applied to compare the results produced by each approach.

4.2 Design Choices

In the development of both PDG approaches, several design choices were made to align with the evaluation criteria of expressivity, reliability, and controllability. Constraints were also introduced to limit the number of parameters and keep the evaluation process focused and manageable.

Fixed Map Size, the dungeon map is designed with a fixed size of 50x50 cells, in line with the CA implementation used in [7]. This size was kept to keep the project manageable while still allowing for enough space to explore the impact of different parameters on the generated dungeons.

Room Size, is a controlled input parameter, with a limited set of predefined configuration to limit the scope of the evaluation. The constraints are set to prevent sprawling layouts, a particular problem for the CA-approach (as seen in Figure 2), and to ensure a fairer evaluation, where BSP and CA are evaluated under somewhat similar spatial constraints. Due to the different nature of the room generation in each method, the size constraints were set slightly differently. BSP naturally imposes size boundaries through recursive spatial subdivision. The size thresholds are defined as: Small Room, minimum 4x4 cells and maximum 8x8 cells. Medium Room, minimum 6x6 cells and maximum 12x12 cells. Large Rooms, minimum 8x8 cells and maximum 16x16 cells. Since Cellular Automata (CA) does not explicitly generate rooms, this thesis introduces two post-processing steps to extract usable room structures: a Flood Fill algorithm to detect contiguous walkable areas, and a custom Room Splitter to divide oversized regions. These steps are not part of the original CA method [7], but are added to adapt the CA output to the evaluation framework. Here the room size is limited by maximum cell count rather than dimensions, the size thresholds are defined as: Small Room, maximum 50 cells, Medium Room, maximum 100 cells. Large Room, maximum 150 cells. No-limit Room, no set max cell count. The CA size limits are set within the BSP room bounds, it is an approximate value to accommodate the spacing offset introduced during the RoomSplitter operations.

Room Offset, a room offset of one tile was introduced in the generation process to prevent rooms from being placed directly adjacent to each other or overlapping. This is especially important in the Binary Space Partitioning (BSP) approach, where rooms are created by recursively subdividing space. Without an offset, rooms may accidentally share walls or corners, leading to visual artifacts, pathfinding errors, or unwanted connections between rooms. By

adding a one-tile margin between rooms and their partition boundaries, it ensures that each room remains spatially isolated. See Figure 1 in Section 3.3 for an example of different offset values.

Start and End Points, the Start Room was assigned randomly, and the End Room the room furthest away from the starting Room. This design decision was made to encourage the player to traverse more of the dungeon, which in turn promotes engagement with the generated dungeon. This approach also aids in evaluating the difficulty and diversity metrics, by allowing a simulated agent using the A* pathfinding agent (see Section 3.6 and 3.7) to navigate from a start and end point.

Room Connectivity, one of the main design goals was to ensure that all rooms are connected to each other, to fulfill the reliability property. This is achieved through the use of a Minimum Spanning Tree in the first step of the corridor generation. This ensures that the dungeon is navigable and does not result in isolated rooms that cannot be accessed. After creating the MST, additional corridors are introduced to add complexity to the dungeon. The number of extra corridors is controlled through two settings: 50% extra corridors and 100% extra corridors. This decision allows for variation in the final dungeon layout. More corridors increase the complexity of the dungeon by introducing alternative paths and potential loops, while fewer corridors result in a more linear and straightforward layout. The extra corridors are strategically placed by checking for potential collisions with existing corridors, ensuring that the dungeon remains coherent and navigable.

A* Pathfinding, the use of the A*-pathfinding algorithm is another important design choice. A* is employed to calculate the most efficient path for connecting rooms, which ensures that the generated corridors do not overlap with obstacles or other rooms. A* is an optimal pathfinding algorithm that provides the shortest path between two points, which is crucial for creating corridors that are both efficient and navigable.

4.3 Room Generation

BSP recursively divides the dungeon map into rectangular sub-regions using randomized vertical or horizontal splits. Splitting continues until sub-regions reach the configured minimum size, corresponding to one of the room size presets: Small (min 4x4, max 8x8), Medium (6x6–12x12), and Large (8x8–16x16). Within each final region, a room is carved out using an offset of 1 cell, to prevent room clipping.

CA builds on the approach described in [7], but with the initial configuration of: Fill Percentage = 50%, Iteration = 3, using a More 3x3 neighborhood definition and a Transition rule of 4. To introduce room structure and size constraints, two post-processing steps are applied: Flood Fill identifies connected clusters of walkable tiles as room candidates (using non-diagonal connections). Room Splitting, recursively divides any oversized rooms into sub-rooms along the longest axis, using a 2 cell offset between them. Room sizes are constrained by total tile count, and uses the room size presents of: Small (max 50 cells), Medium (max 100 cells), Large(max 150 cells) and Nolimit (no max cell count).

4.4 Corridor Generation

After generating rooms, corridors are constructed to ensure all rooms are reachable. This process has two main steps: building a Minimum Spanning Tree (MST) and then creating additional corridors to improve navigability (see Figure 3 and 4).

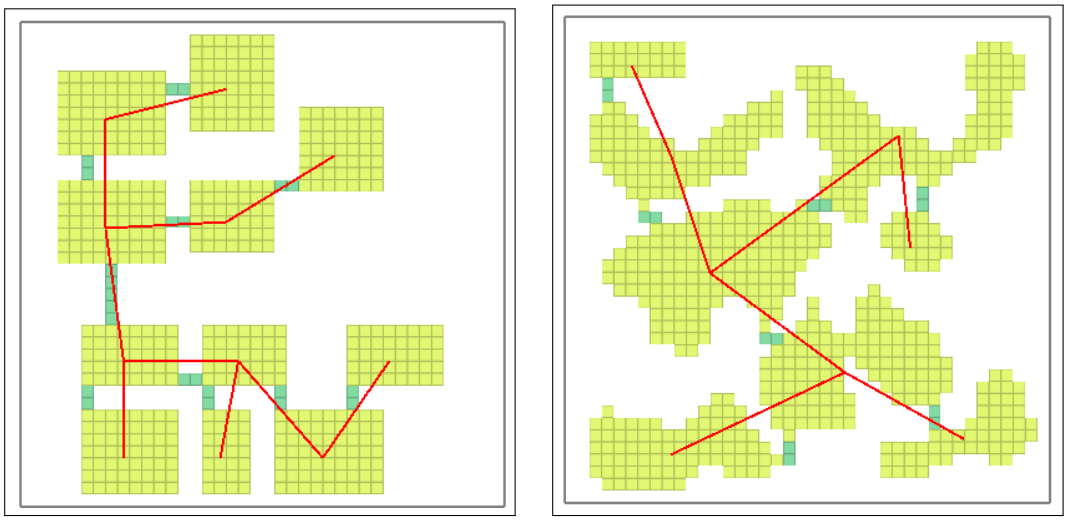
MST Construction

The MST ensures that all rooms are connected using the shortest total corridor length, avoiding cycles and isolated components. The room layout is first modeled as an undirected graph where each room is a node and edges represent potential connections (see Section 3.5 for the graph representation and Prim's algorithm pseudocode).

The MST is constructed through the following steps:

1. Starting with a randomly selected room as the first node.
2. Repeatedly selecting the shortest connection (based on tile distance) between rooms already in the tree and those outside it.
3. Adding a corridor between the closest rooms and updating the tree until all rooms are connected.

Each connection is translated into a spatial corridor using the A* pathfinding algorithm (see Section 3.6), which computes a grid-based path while treating rooms and previously placed corridors as obstacles. Figure 3 shows this process. The rooms and their MST connections correspond to a tree structure: each room is a node, and each corridor from the MST is an edge.



(A) (B)

Figure 3: Show dungeon layouts after the MST has been constructed. The red lines represent connections by the MST and the green tiles represent the built corridors. A) Show the BSP layout. B) Show the CA layout.

Additional Corridors

While the Minimum Spanning Tree guarantees connectivity between all rooms, it produces a tree structure with no cycles. This results in linear dungeon layouts with only one possible path between any two rooms. To introduce complexity, additional corridors can be added on top of the MST to form loops and alternative routes.

The Additional Corridor Algorithm (see Algorithm 5), is custom method designed to add extra corridors while avoiding collisions and discarding connections that is difficult to translate into valid spatial corridors. By using direct lines between room centers as preliminary

candidates and obstacles, the algorithm can filter out infeasible options early, especially when insufficient geometric space exists for the A* pathfinding algorithm to produce a clean path (see Section 3.6).

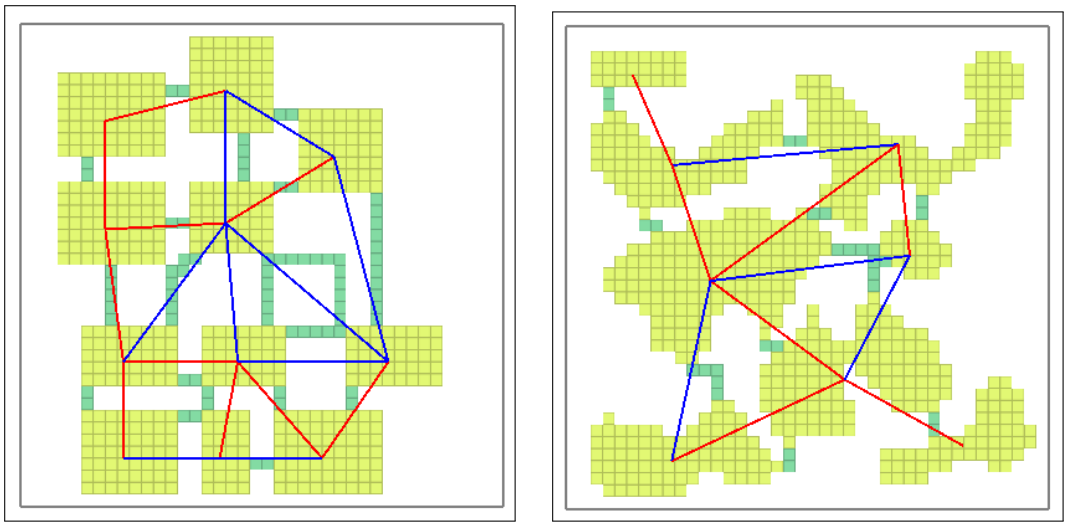
The algorithm operates in two stages: candidate selection and corridor building. In the first stage, valid room pairs not already connected by the MST are identified. For each pair, a straight line is drawn between the rooms' center points. This candidate is then evaluated: it is discarded if the line intersects an existing connection (excluding shared endpoints), crosses a placed corridor, or passes through any third room. To maintain spacing, a buffer of one tile is added around all other rooms.

Algorithm 5: Additional Corridor Generation

Input: Graph G (with rooms and MST corridors), Offset o (collision buffer)
Output: List of additional corridors with valid tile paths

$C_{\text{new}} \leftarrow$ empty list of corridors;
$L_{\text{existing}} \leftarrow$ list of center lines from existing MST corridors;
$P_{\text{pairs}} \leftarrow$ unconnected room pairs in G;
foreach *room pair* (A, B) *in* P_{pairs} **do**
    $line \leftarrow$ straight line between centers of A and B;
    $obstacles \leftarrow$ all tiles from other rooms (with o-tile buffer);
    if *line intersects any in* $L_{existing}$ ***or*** *line intersects* $obstacles$ **then**
        skip this pair;
    else
        Add $line$ to L_{existing};
        C_{new}.add(Corridor(A, B));
$T_{\text{blocked}} \leftarrow$ tiles of all rooms and existing corridors;
foreach *corridor in* C_{new} **do**
    $(start, end) \leftarrow$ closest adjacent tiles between the two rooms;
    Temporarily remove $start$, end from T_{blocked};
    $path \leftarrow$ A*$(start, end, T_{\text{blocked}})$;
    if $path$ *is not empty* **then**
        Add $path$ to T_{blocked};
        Set corridor path, start, and end tiles;
return C_{new};

If the connection passes all checks, it is accepted. In the second stage, a valid tile-based path is generated using A* pathfinding (see Section 3.6). Obstacles include all other rooms, corridors, and their walls—except the entrance and exit tiles, which are temporarily excluded. If a path is found, it is added to the global obstacle map, and the process continues until the desired number of additional corridors is created. This is defined as 50% or 100% of all valid room pairs (rounded up).



(A) (B)

Figure 4: Show dungeon layouts after the MST has been constructed and Algorithm 5 has created additional corridors. The red lines represent connections by the MST, the blue lines the extra connections by Algorithm 5 and the green tiles represent the build corridors and extra corridors. A) Show the BSP layout. B) Show the CA layout.

4.5 Experiment Setup

To evaluate the performance of the BSP and CA generation methods, a series of controlled experiments were conducted using multiple input configurations. The project was implemented in C# within the Unity game engine (version 2022.3.26f1), using a 2D grid-based environment with a fixed map size of 50×50 tiles.

Two separate dungeon generators were developed: one using BSP and one using CA. Each generator follows a two-step process. First, rooms are generated according to the selected method and room size parameters. Second, corridors are added to connect the rooms, resulting in a complete and navigable dungeon layout. These layouts are then evaluated using the agent-based metrics Difficulty and Diversity (see Section 4.7).

Each test run is defined by a specific configuration. A configuration consists of a combination of:

- **Generation Method:** BSP or CA
- **Room Size:** BSP (Small, Medium or Large), CA (Small, Medium, Large or NoLimit).
- **Corridor Density:** 50% or 100% of all valid, non-colliding unconnected room pairs (rounded up)

Each configuration is executed 1000 times, meaning that 1000 dungeon layouts are generated, evaluated, and logged per configuration. This ensures statistical reliability and provides a dataset large enough for meaningful comparison. In total, 14,000 dungeon instances were generated. The full list of tested configurations is as follows:

BSP configurations (6 total): BSP-Small-50%, BSP-Small-100%, BSP-Medium-50%, BSP-Medium-100%, BSP-Large-50% and BSP-Large-100%.

CA configurations (8 total): CA-Small-50%, CA-Small-100%, CA-Medium-50%, CA-Medium-100%, CA-Large-50%, CA-Large-100%, CA-NoLimit-50% and CA-NoLimit-100%.

The total number of configurations differs between BSP and CA because the CA generator includes an additional NoLimit setting, which does not enforce a maximum room size. This variant is to check the Room Splitting post-processing step, how CA perform with or without a set room size constraint.

4.6 Evaluation

Each of the 14 input configurations (6 BSP and 8 CA), was run 1000 times, generating a total of 14'000 dungeon instances. These instances were then evaluated through an agent-based simulation, where an agent was tasked with navigating from the start room to the end room. After each simulation run, the two key evaluation metrics, Difficulty and Diversity, were computed [2]. (see Section 3.7 for a detailed description)

$$\text{Difficulty} = \frac{\text{ExploredTiles}}{\text{TotalReachableTiles}}$$

$$\text{Diversity}(s_i) = \frac{1}{5} \sum_{j=1}^{5} \text{Levenshtein}(s_i, s_j) \quad \text{for } i \neq j$$

The resulting metric values, ConfigurationID, DungeonID, Difficulty, and Diversity, were stored in a CSV file. Before visualization, all metric values were normalized to fall within the range [0, 1]. (see Figure 5)

```
ConfigID,DungeonID,Diversity,Difficulty
BSP-small-50,0,0.68,0.22
BSP-small-50,1,0.79,0.63
BSP-small-50,2,0.71,0.60
BSP-small-50,3,0.87,0.35
BSP-small-50,4,0.90,0.19
BSP-small-50,5,0.72,0.39
BSP-small-50,6,0.68,0.64
BSP-small-50,7,0.85,0.71
BSP-small-50,8,0.87,0.30
BSP-small-50,9,0.77,0.40
```

Figure 5: Shows a sample of the resulting CSV file, with normalized Difficulty and Diversity scores.

To analyze the Expressive Range of the two generation approaches, the results were aggregated for each configuration. The normalized Difficulty and Diversity values for all 1000 dungeon instances per configuration were visualized as a 2D density heatmap, plotted on a 10×10 grid. (see Figure 6)

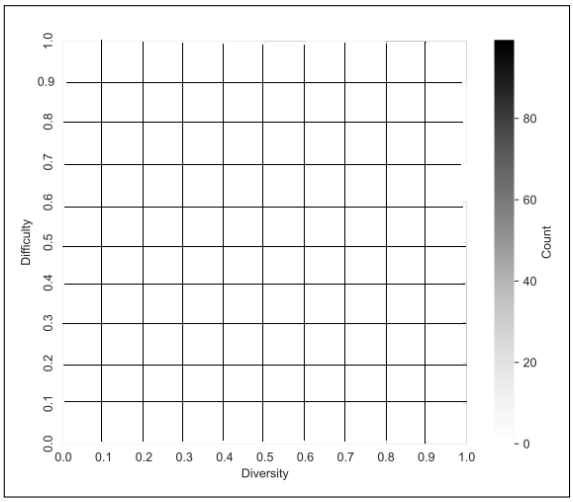


Figure 6: Shows a 10x10 heatmap, with the diversity score as the x-axis and the difficulty score as the y-axis. Each dungeon layout's metrics are plotted on the grid, the darker a section appears on the heatmap, the higher the count of layouts with those metrics.

4.7 PDG Taxonomy

Defining this thesis's two approaches according to the taxonomy as defined by [17], and originally developed by [15], gives the following categorization:

- Content Need: required, the dungeon layout is an essential feature that is required for player navigation.
- Generation Time: offline, the dungeons are generated during game setup or before use.
- Generation Control: moderate, the BSP provides controllability through room size thresholds, while CA allows initial grid seed control as well as room splitting. BSP offers slightly more predictable control, while CA introduces more emergent behavior from fewer parameters.
- Generality: general use, the PDG output is not adapted based on player behavior of preferences.
- Random Choice: stochastic, both approaches employ stochastic processes. CA's evolution rules introduce randomness in cell updates, while BSP incorporates random splits and room placements. Neither method is deterministic, identical inputs may yield different outputs.
- Generation Method: both BSP and CA are constructive methods, generating content in a single pass without iterative refinement. However, CA may require additional post-processing to segment viable rooms, which introduces light generate-and-test elements.
- Content Authorship: fully Automated, designers influence outputs only through parameter settings; no mixed-initiative features are used in this implementation.

5 Results

5.1 Visual Output of Dungeon Layouts

Running the experiment setup, 1000 dungeon layouts were created for each of the 14 configurations, 6 for the BSP and 8 for the CA approach. Figure 7 shows a generated sample dungeon for each of the BSP configurations and Figure 8 shows a generated sample for each of the CA configurations. Visualizing a sample output shows some typical output characteristics.

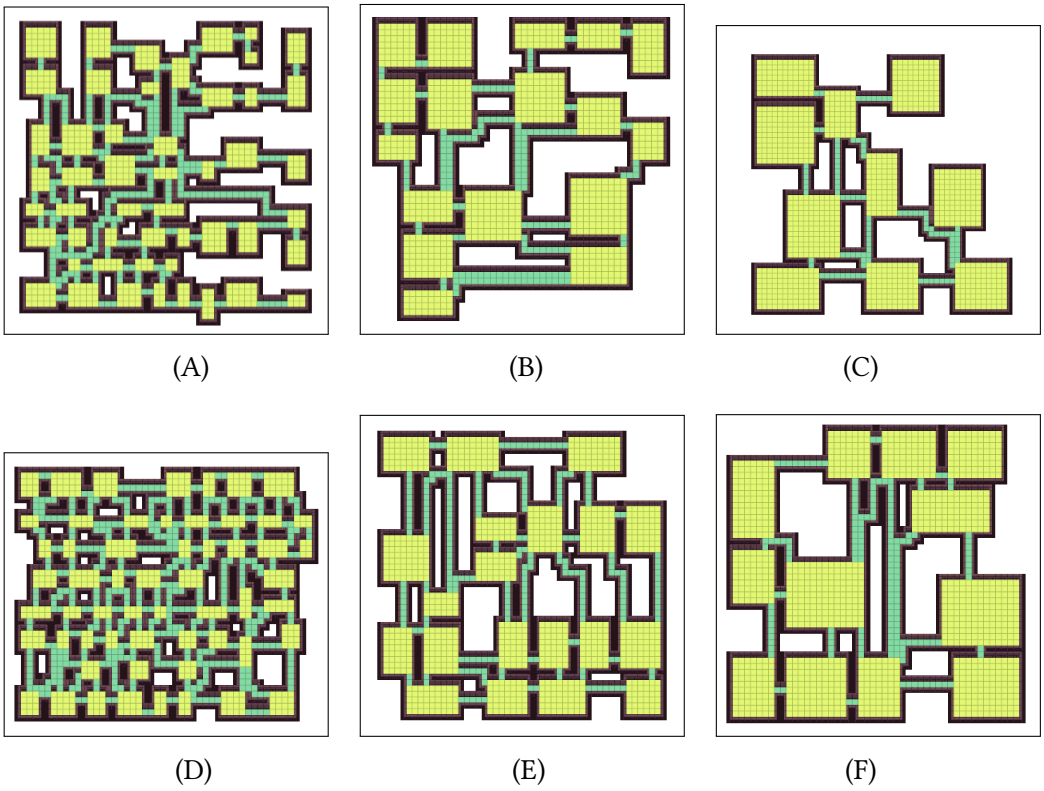


Figure 7: Shows a sample BSP generated dungeon of each input configuration: A) BSP-small-50, B) BSP-medium-50, C) BSP-large-50, D) BSP-small-100, E) BSP-medium-100, F) BSP-large-100.

Figure 7 shows that all BSP configurations generate dungeon layouts in which all rooms appear reachable. The corridor generation creates many navigable paths, and in layouts A, B, C, D, and F, corridors visually merge to form broader passageways. Layouts A and D, which use the ”small” room size, appear visually cluttered. In contrast, layouts B, D, E, and F present a clearer visual structure. The configurations with 100% extra corridors (D, E, and F) show a notable increase in the number of paths, enhancing navigability but also introducing visual clutter and excessive connectivity.

Across all samples, there are no visual artifacts such as disconnected tiles or malformed rooms. While room placement appears well-balanced across the map, many rooms—especially in layouts A and D—are aligned vertically or horizontally with their neighbors, creating an

artificial, grid-like appearance.

Figure 8 shows the CA configurations, and that some rooms in (Layout A and C) appear to be unreachable, indicating reliability issues (see Section 6). Layout D has some unusual pathing, with corridors that twist and wind awkwardly around rooms. Layout G and H illustrate the problem of large, open spaces, a key issue that the room-splitting component is designed to address. Several of the more complex CA configurations also show minor visual artifacts, such as isolated tile gaps, which may point to inconsistencies in generation reliability. Despite this, the CA layouts with the room divider (A-F) manage to preserve the organic, cave-like feel seen in the default CA maps (G and H) while introducing more clearly defined and structured room spaces.

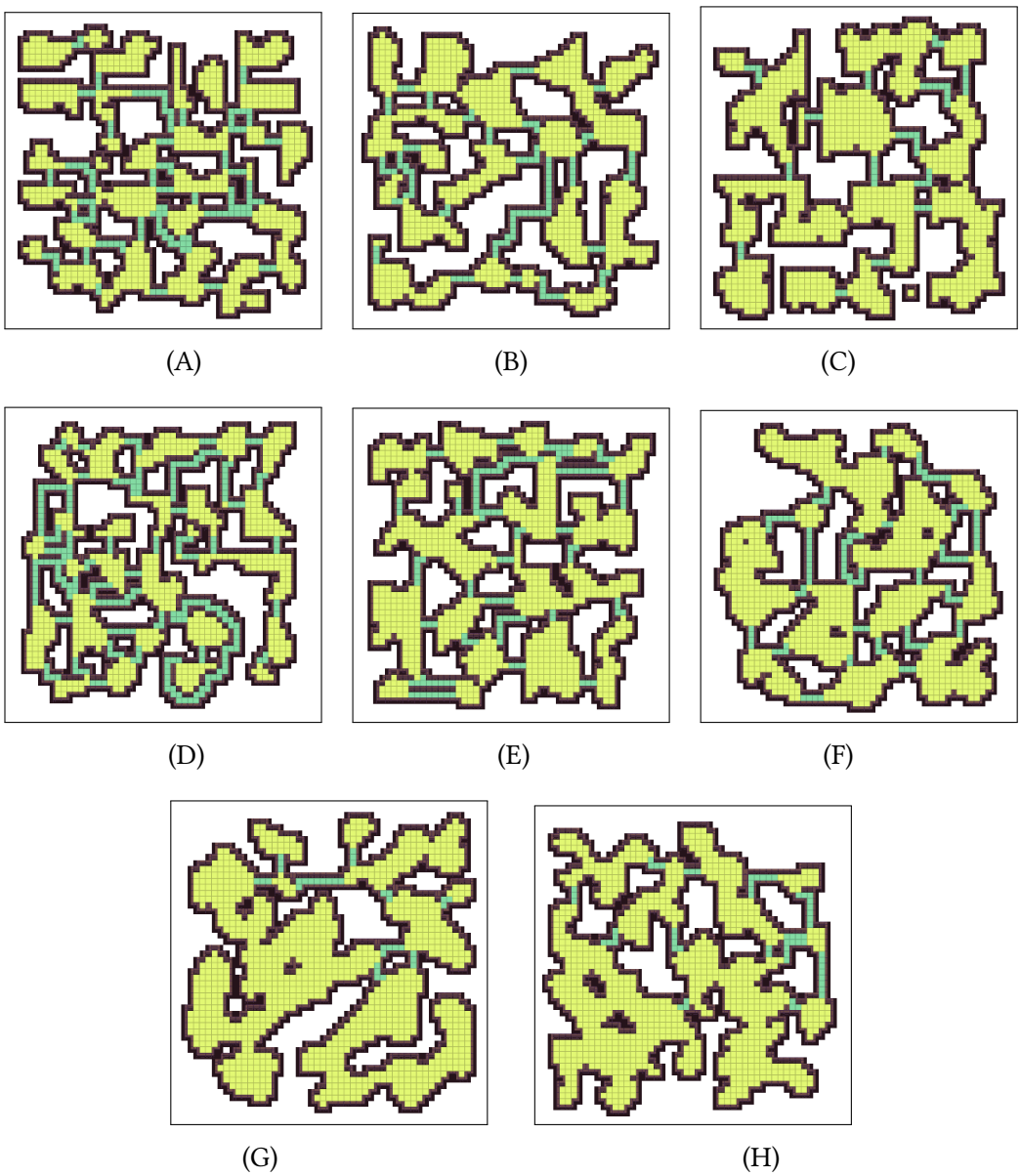


Figure 8: shows a sample CA generated dungeon of each input configuration: A) CA-small-50, B) CA-medium-50, C) CA-large-50, D) CA-small-100, E) CA-medium-100, F) CA-large-100, G) CA-nolimit-50, H) CA-nolimit-100.

5.2 Expressive Range via Difficulty and Diversity

Figure 9 to 12 present heatmaps of the expressive range for each configuration, plotting the generated dungeon layouts by their measured Difficulty and Diversity. Each heatmap reflects the distribution of 1000 generated samples over a 10x10 grid space, to observe general trends as the input parameters change.

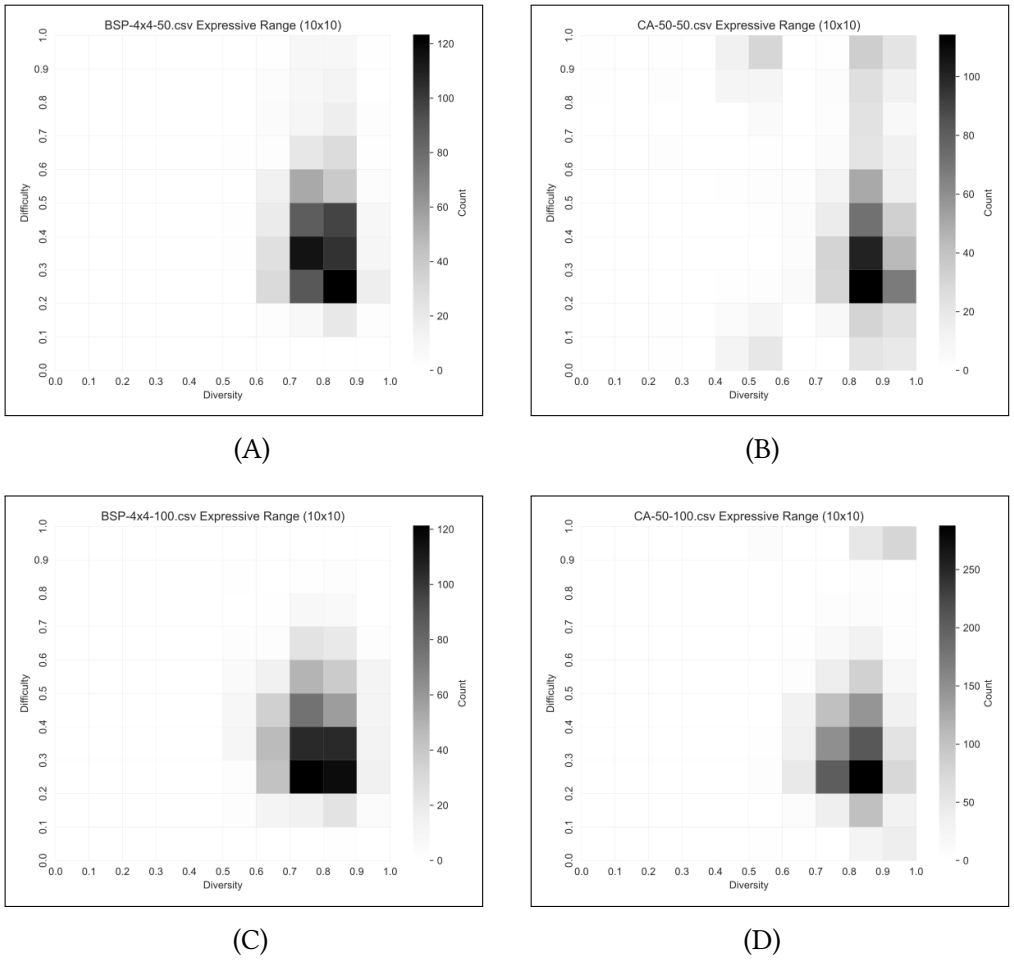


Figure 9: Heatmap for the four configurations with the room size Small. The x-axis represent the Diversity score, and the y-axis the Difficulty score. Darker regions indicate a higher count of dungeon layouts with that score combination. A) BSP-small-50, B) CA-small-50, C) BSP-small-100, D) CA-small-100.

Figure 9 displays small rooms, Figure 10 displays medium rooms, Figure 11 displays large rooms. A and B (in Figure 9,10 and 11) shows 50% extra corridors, with C and D (in Figure 9,10 and 11) showing 100% extra corridors. Figure 12 show CA configurations with no size limit.

BSP is consistent across input configurations, displaying a tight cluster with high diversity and low to medium difficulty. Some BSP configurations with 50% extra corridors, such as Figure 9A (BSP-small-50) and Figure 10A (BSP-medium-50), show a higher expressive range in terms of difficulty compared to configurations like Figure 9C (BSP-small-100) and Figure 10C (BSP-medium-100), which use 100% extra corridors. Overall, for BSP, room size does not appear to significantly impact the distribution of Difficulty and Diversity.

CA, on the other hand, exhibits a broader expressive range. The layouts generated using CA are generally more spread out than those from BSP. Like BSP, CA configurations also form

clusters with high diversity and low to medium Difficulty, visible in Figure 9B, 10B, and 11C. In addition to these main clusters, CA configurations consistently produce smaller, distinct clusters, referred to as island clusters, in three regions: one with high diversity and high difficulty (top right), one with medium diversity and high difficulty (top center), and one with medium diversity and low difficulty (bottom center). These island clusters are particularly noticeable in the 50% corridor configurations, such as Figure 9B, 10B, and 11B.

When comparing CA configurations with 50% corridors to those with 100% corridors (e.g., Figure 9D, 10D, 11D), the island clusters become less prominent in the latter. Additionally, the size of the rooms appears to influence the density and clarity of these island clusters, with larger room sizes (e.g., Figure 11B and 11D) showing more pronounced island clustering compared to smaller ones.

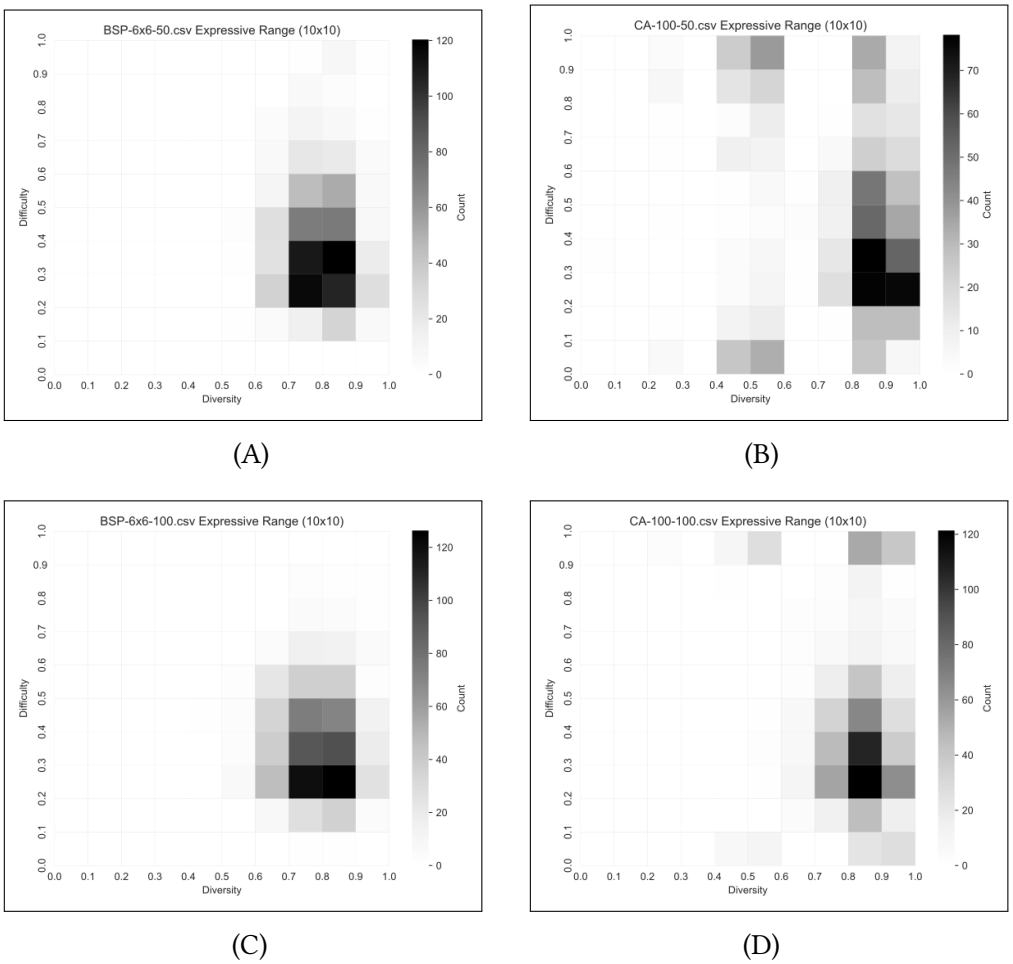
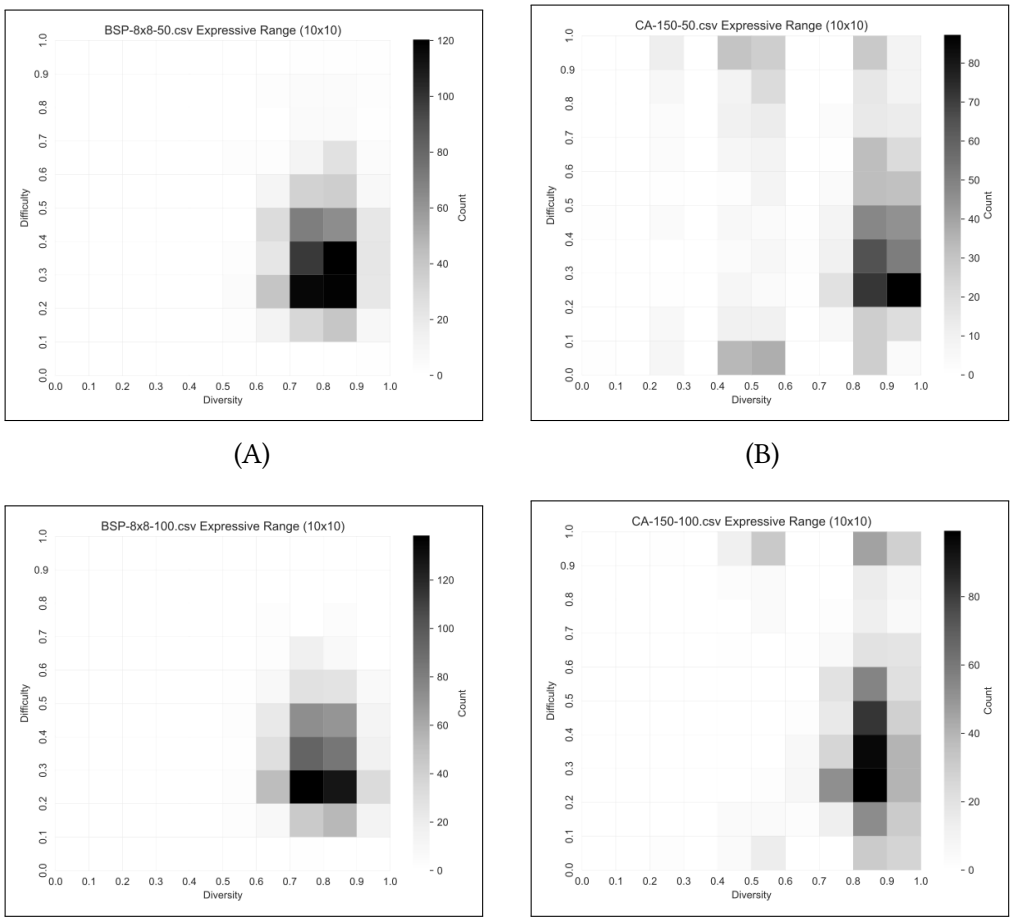


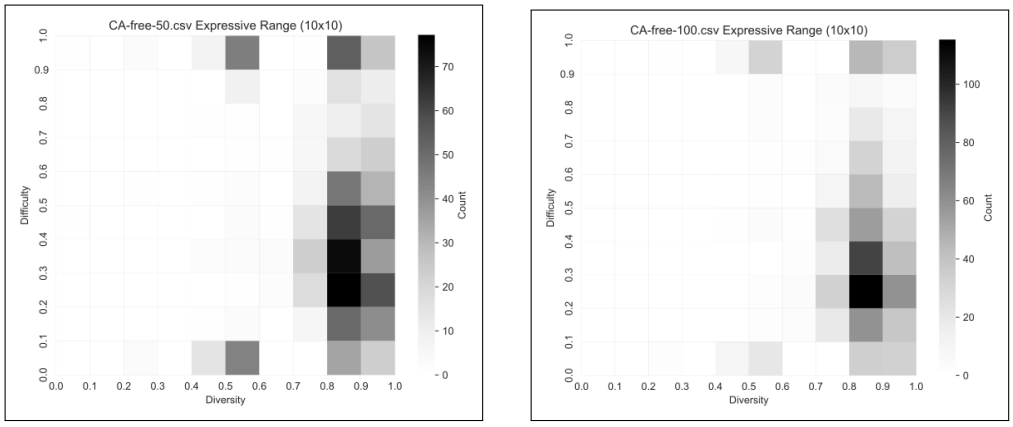
Figure 10: Heatmap for the four configurations with the room size Medium. The x-axis represent the Diversity score, and the y-axis the Difficulty score. Darker regions indicate a higher count of dungeon layouts with that score combination. A) BSP-medium-50, B) CA-medium-50, C) BSP-medium-100, D) CA-medium-100.



(A) (B)

(C) (D)

Figure 11: Heatmap for the four configurations with the room size Large. The x-axis represent the Diversity score, and the y-axis the Difficulty score. Darker regions indicate a higher count of dungeon layouts with that score combination. A) BSP-large-50, B) CA-large-50, C) BSP-large-100, D) CA-large-100.



(A) (B)

Figure 12: Heatmap for the two CA configurations with no size constraints. The x-axis represent the Diversity score, and the y-axis the Difficulty score. Darker regions indicate a higher count of dungeon layouts with that score combination. A) CA-nolimit-50, B) CA-nolimit-100.

6 Discussion

What makes a "good" dungeon generator? This thesis examined three desirable properties: expressivity, reliability and controllability [13] of CA and BSP.

A larger part of this thesis examined the expressivity aspect, through an expressive range analysis on the agent based metrics as proposed by [2]: diversity and difficulty. The results from Figure 9 through 12 show that CA has a larger expressive range than BSP with more clusters and was more susceptible to input changes. The BSP approach created a more rigid and predictable layout. CA had some smaller island clusters that can be hard to interpret as the visual examination of the sample layouts showed reliability issues, which will be discussed below. These islands seem to be "extreme" in the sense that they have either maximum or minimum difficulty. These results indicate that when choosing between the two approaches with expressivity in mind the CA approach may be more varied and sporadic. This echoes the discussion in [16] about CA's ability to create chaotic maps.

In terms of reliability this thesis's focus was to ensure a minimum connectivity between rooms. BSP was highly reliable throughout, as shown in Section 5.1, whereas CA was more inconsistent, creating spatial artifacts and isolated rooms. BSP created rooms and corridors as expected with no hurdles while the CA approach ended up creating some long winded paths. If high reliability is the priority then BSP should be the given choice.

CA by default has low controllability. This thesis proposed the use of a room splitter to guide the room generation process and to give the designer the ability to set max room sizes. BSP on the other hand has some controllability but it mostly guides the process rather than detailed design choices. BSP has slightly more controllability when compared to CA using the aid of some post processing such as the proposed room splitter.

Several limitations should be acknowledged with the thesis. First the evaluation relies heavily on the two agent based metrics Difficulty and Diversity, which ended up being hard to draw conclusions from. The range of the input configuration was also a limiting factor of the study, but it was done to reduce the scope of the evaluation. The reliability issues with the CA approach was not solved, but could perhaps be an issue with the algorithm implementation used in this thesis.

7 Conclusion

This thesis compared two constructive PDG approaches, Binary Space Partitioning (BSP) and Cellular Automata (CA), through the lens of expressivity, reliability, and controllability. Results show that BSP is reliable and predictable, producing structurally sound layouts with some control over output, but limited expressive variety. CA, by contrast, showed a broader expressive range, with more varied and sometimes extreme outputs, but also less reliability, occasionally generating unreachable rooms or artifacts. While BSP suits use cases requiring consistency and control, CA may better serve designs prioritizing variety and emergent structure, though it might benefit from additional constraints like room splitting.

The choices between BSP and CA depends on design priorities: BSP is preferable for structured and predictable layouts, while CA might be better for scenarios demanding variety and emergent structure.

References

[1] J. R. Baron. Procedural dungeon generation analysis and adaptation. In *Proceedings of the Southeast Conference ACM SE'17*, pages 168–171, New York, NY, USA, 2017. ACM.

[2] M. Beukman, S. James, and C. Cleghorn. Towards objective metrics for procedurally generated video game levels. arXiv preprint arXiv:2201.10334, 2022.

[3] J. A. Brown and M. Scirea. Procedural generation for tabletop games: User driven approaches with restrictions on computational resources. In *Proc. 6th Int. Conf. Software Engineering for Defence Applications (SEDA)*, pages 44–54, Rome, Italy, 2018.

[4] M. C. Green et al. Two-step constructive approaches for dungeon generation. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, pages 1–7, New York, NY, USA, 2019. ACM.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] V. Jarník. O jistém problému minimálním [about a certain minimal problem]. *Práce Moravské Přírodovědecké Společnosti*, 6(4):57–63, 1930. in Czech.

[7] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, New York, NY, USA, 2010. ACM.

[8] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, February 1966.

[9] Y. Li, Z. Wang, Q. Zhang, B. Yuan, and J. Liu. Measuring diversity of game scenarios. *IEEE Transactions on Games*, pages 1–29, 2025.

[10] J. Liu, S. Snodgrass, A. Khalifa, S. Risi, G. N. Yannakakis, and J. Togelius. Deep learning for procedural content generation. *Neural Computing & Applications*, 33(1):19–37, 2021.

[11] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, November 1957.

[12] R. A. Schumacker, B. Brand, M. G. Gilliland, and W. H. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, 1969.

[13] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Springer International Publishing, Cham, softcover reprint of the original 1st edition 2016 edition, 2018.

[14] T. Smith et al. Graph-based generation of action-adventure dungeon levels using answer set programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, pages 1–10, New York, NY, USA, 2018. ACM.

[15] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[16] R. van der Linden, R. Lopes, and R. Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2014.

[17] B. M. F. Viana and S. R. dos Santos. A survey of procedural dungeon generation. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 29–38, New York, 2019. IEEE.

[18] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

UMEÅ UNIVERSITY