Universiteit Leiden
The Netherlands

Computer Science

Foundations of Computing

Monte Carlo Tree Search-driven Multisolution Puzzle Level Generation

Perri van den Berg

Supervisors:
Mike Preuss & Matthias Müller-Brockhausen

MASTER THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

16/06/2025

Abstract

Procedural Content Generation (PCG) often falls short of capturing the structure and nuance of human-designed game levels. We argue that more human-like levels can be achieved by optimizing for specific design qualities such as multiple solution paths, structural coherence, and diversity. To this end, we propose a level generation method based on Monte Carlo Tree Search (MCTS), aimed at producing levels with meaningful structure and multiple solution paths. We evaluate our approach across a large set of generated levels using three metrics: overall quality, uniqueness relative to previously generated levels, and the proportion of imperfect levels.

To assess the human-likeness of the generated levels, we applied our method to a prototype game, *Rescue Elite*, inspired by the Commodore 64 classic *Fort Apocalypse*. Through a user study with 21 participants, we collected gameplay data (performance and frustration markers) to evaluate level flow and difficulty curve, and conducted a follow-up survey. Notably, none of the participants realized the levels were AI-generated. We also collected additional data, such as solution paths, damage markers, and paths that led to player deaths, which we analyzed. Overall, MCTS guided by our constraints is a viable approach for generating human-like PCG game levels.

Contents

1 Introduction

In recent years, the roguelike game genre has gained significant popularity, inspiring many games to adopt Procedural Content Generation (PCG) for creating levels, weapons, and even entire worlds [1]. This approach enables seemingly endless content, as seen in games like *No Man's Sky*, *Bad North*, and *Balatro*. However, procedurally generated content often feels either too random or overly constrained, lacking the depth and thoughtful design typically associated with handcrafted levels. As a result, many games rely heavily on their core gameplay mechanics while treating procedural generated content as a secondary feature.

One of the key challenges in PCG is managing the tradeoff between control and variability. Handcrafted content allows designers to better curate player experiences, introducing challenges, surprises, and emotional impact that enhance the flow of the game. However, it is time consuming and limited in replayability. In contrast, procedurally generated content supports high replayability but often lacks these carefully designed elements [2]. Though, some games attempt to incorporate handcrafted-like features into procedural systems (e.g., scripted events or puzzle structures), the novelty of these features tends to fade after repeated encounters.

In *Dead Cells*, for instance, the developers use a hybrid approach that combines procedurally generated level layouts with handcrafted room modules [3]. This strategy ensures consistent pacing and flow while still offering variation between runs. Nevertheless, players eventually begin to recognize recurring room patterns, which can reduce the sense of discovery over time. Similarly, *Spelunky* assembles its levels from randomized traps and enemy placements. Although this produces emergent gameplay, experienced players eventually learn to anticipate common configurations, reducing the impact of surprise and exploration.

Despite these limitations, PCG offers several important advantages. Most notably, it supports the generation of new content with each playthrough, helping maintain player interest and challenge. In roguelike games such as *Rogue*, unpredictability is central to the gameplay loop, as players must navigate unfamiliar dungeons every session [4]. Likewise, games like *Blue Prince* and *Tetris* use procedural elements to create variation and replayability across sessions. However, as [2] notes, procedural systems often fall short of the design coherence and polish found in handcrafted levels.

To bridge the gap between handcrafted and procedurally generated levels, we explore the use of Monte Carlo Tree Search (MCTS) [5] to create structured, engaging level descriptions. We first detail our MCTS implementation (Section 3), define what constitutes a "good" level (Section 3.3), and outline which parameters can be configured for level generation (Section 5.1). One of our key criteria is replayability. By applying MCTS to this definition, we generate levels as graph-based representations, where nodes denote rooms and edges represent doors.

To further evaluate the effectiveness of our MCTS-generated level descriptions, we developed our prototype game *Rescue Elite* (Section 4.1), inspired by the Commodore 64 title *Fort Apocalypse*. In this game, the abstract graph-based level descriptions produced by MCTS are converted into playable levels using various PCG techniques (e.g., Cellular Automata). To evaluate these AI-generated levels, we conducted a user study with participants experienced in gaming (Section 5.2). After playing the levels, participants completed a survey. This study assesses game's flow, level structure, and overall player satisfaction to determine whether our approach successfully mimics the design qualities of handcrafted levels.

In Section 2, we review prior work on MCTS, procedural level generation algorithms, and methods for designing games with multiple solutions paths. Section 3 details our MCTS-based level generation

pipeline and evaluation metrics. In Section 4, we describe the PCG methods used to generate playable levels from abstract graphs. Section 5 presents our experimental setup and key findings from both MCTS evaluations and user feedback. We conclude in Section 7 with a summary, limitations, and suggestions for future research.

2 Related work

In this section, we review the use of Monte Carlo Tree Search (MCTS) in game design and puzzle solving and how we can evaluate level layouts. Furthermore, we discuss various Procedural Content Generation (PCG) techniques for level generation, and survey approaches aimed at supporting multiple solutions in game design.

2.1 MCTS for Game Design and Puzzle Solving

In our research, we employ Monte Carlo Tree Search (MCTS) [6], a decision-making algorithm that builds a search tree over multiple iterations while balancing exploration (trying new possibilities) and exploitation (refining known good paths). MCTS has proven effective in a wide range of applications, particularly in creative domains [5].
One example of MCTS is in narrative generation. In [7], MCTS is used to generate murder mystery storylines by simulating character interactions within a domain-specific rule set. Through user studies, the authors found that MCTS-generated stories were perceived as more enjoyable compared to those generated by random methods. This closely relates to our approach, in which MCTS operates under a structured rule set to generate logical and coherent level graphs.
Another use case is level generation for the puzzle game *Angry Birds*, described in [8], where MCTS is combined with AI agents for evaluation. The performance of agents on generated levels influences the scoring mechanism during the evaluation step of MCTS. We adopt a similar idea by evaluating the generated level graphs using simple AI agents to measure how a human player would perform.

2.2 Evaluation Methods in Level Design

Evaluating the quality of procedurally generated levels often requires more than visual inspection or simple playtesting. Several studies have proposed quantitative metrics to measure level design in a structured and repeatable way. For example, [9] present a framework for analyzing platformer levels using measurable properties such as linearity and leniency. These metrics provide a mathematical basis for comparing different levels and capturing design qualities like challenge, pacing, and openness.
[10] makes use of genetic algorithms in which they define a fitness function which, using heuristics, evaluates a level. They make use of 5 evaluation criteria: how the path is structured, the meaning of the individual tiles of the grid level, the ending of the level has to be both possible and challenging, the aesthetics of the level, and the usage of the space in the level. Using these well-defined properties, the authors improved the output quality in a systematic way.
Inspired by these approaches, we evaluate the levels generated by our MCTS-based method using three metrics: overall quality, uniqueness relative to previously generated levels, and the proportion of imperfect levels (e.g., unplayable or trivially solvable). These metrics provide concrete feedback that helps us assess and refine the performance of our generative system.

2.3 PCG Methods for Level and Puzzle Generation

Procedural Content Generation (PCG) refers to algorithmic techniques for creating game content automatically, such as levels, terrain, or missions [11]. These methods are commonly used to increase

replayability, reduce manual design effort, and introduce variability into games.
To transform our graph-based level descriptions into 2D playable layouts, we use PCG techniques such as Cellular Automata (CA). The work of [12] demonstrates how multi-layered CA can generate island-like levels that look organic and coherent. We adopt similar techniques for generating the cave-like layouts of our levels.
We also draw on ideas from procedural puzzle generation. For example, [13] surveys puzzle generation techniques that allow for varied difficulty and multiple possible solutions. Since our system prioritizes replayability and structural richness, supporting multiple solutions per level is essential.
Other approaches, such as the Procedural Zelda framework [14], focus on creating exploration-based 2D levels using PCG while considering player experience and engagement. Grammar based methods [15] provide another structured approach to generating levels. Grammar based level generative systems use predefined production rules, similar to those in formal grammars, to iteratively construct game levels with structured and coherent layouts. However, grammar based systems require well defined structures, which can be difficult to combine with the flexible outputs of our MCTS-generated graphs.

2.4 Designing for Multiple Solutions in Games

Designing puzzles with multiple valid solutions improves replayability and encourages diverse player strategies. In [16], the authors explore how puzzles can be designed to allow for multiple solution paths. While they found no universal method for solving 2D puzzles, they outline several strategies that can help solve such puzzles more efficiently. They also examine puzzle complexity using metrics like solution length, branching factor, and the effort required by solvers. In our work, we aim to generate level graphs that similarly support multiple paths, adding both depth and variety to gameplay. However, creating such levels makes the generation process more complex. It becomes more difficult to balance structure, solvability, and meaningful variation.
A contrasting perspective is offered by [17], where the authors investigate what happens when constraints are relaxed during level generation. Their findings show that removing too many constraints can lead to unplayable or trivial levels. This directly affects how we define “good” levels and assess their structural quality in our MCTS-based generation system.

3 Monte Carlo Tree Search

This section details the Monte Carlo Tree Search (MCTS) pipeline used in our experiments, as illustrated in Figure 1. The pipeline is designed to iteratively improve the structure of a randomly generated puzzle level until it meets a specified quality threshold. This is conceptually similar to how diffusion models refine noise into coherent outputs through repeated steps. While our approach is search-based rather than learned, both involve a gradual improvement process guided by an objective.

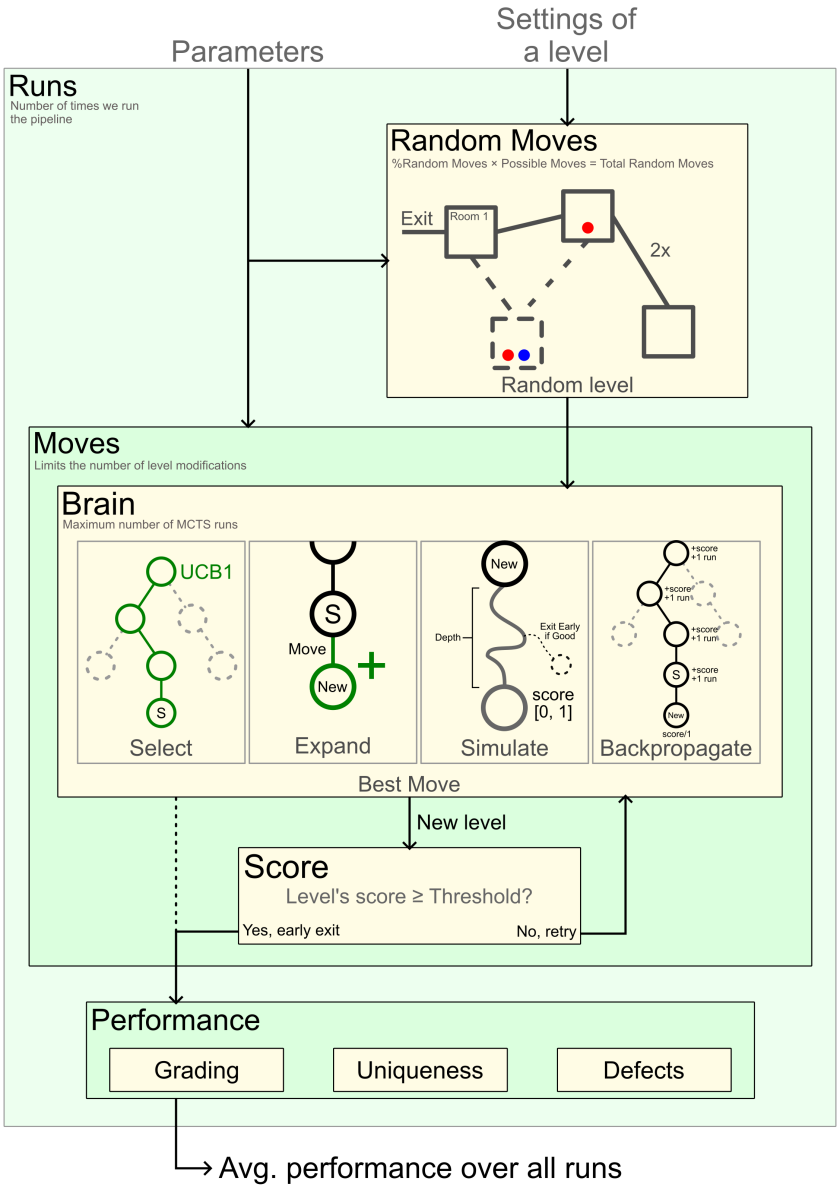

Figure 1: Diagram of the MCTS pipeline used to generate and improve puzzle levels. The pipeline consists of 3 parts: Generating a random level, improving it using MCTS, and evaluating MCTS's output. Using this method over many iterations, we can measure MCTS's performance.

The MCTS pipeline consists of three main stages. In the first stage, a base level is created by applying a series of random modifications (moves) to an initially empty level, a level consisting only of rooms. These random moves involve either adding or removing a placed button or a connection between two rooms. In the second stage, the MCTS algorithm is used to refine the randomly generated level over a fixed number of iterations. During each iteration, MCTS constructs a tree structure (referred to as the "Brain") and returns the best found move. After applying each move, the system checks whether the current level meets the desired quality threshold. If it does, the second stage is exited early. In the final stage, the resulting level is evaluated across multiple runs to compute average performance. The evaluation is based on defined grading criteria, the uniqueness of the level compared to previously generated ones, and the presence of defects (e.g., unsolvable configurations).
The entire process is parameterized by the number of runs (how many times the full pipeline is executed), the number of allowed moves, and the level settings (such as the number of rooms and unique button types). The pipeline outputs the average performance over all runs.

3.1 The MCTS Algorithm

We made several changes to the 'basic' form of Monte Carlo Tree Search, as described in [6], to better suit our domain. The algorithm incrementally builds up a tree structure over a number of iterations starting with the root node. The level stored in the root node is our randomly generated level which was given as the input to the agent, and this level may or may not be playable. During the iterations, MCTS aims to grow the tree towards a level in the search space which is almost perfect according to our evaluation criteria.
The search space consists of all the possible levels that can be generated by applying any number of moves to the root level.
In our representation, level features (connections and buttons) can be toggled on or off, allowing any level to be transformed into another level, which has to the same constraints, by applying a sequence of moves. The total number of possible moves (i.e., the number of toggleable features) is given by:

$$\text{total number of moves } = \frac{1}{2}n(n-1)(b+1) + (b \times n)$$

Here, n is the number of rooms and b is the number of unique button IDs. The first term represents all possible connections between pairs of rooms. A connection may either require a specific button ID to open or have no button requirement. While multiple connections can exist between the same pair of rooms, each must be unique. The second term accounts for all possible button placements. Each room can contain up to b unique buttons, but if there are multiple rooms, Room 1 is not allowed to contain the button which is required to complete the level.
During the set number of 'thinking' iterations of MCTS, we start by selecting the most promising node in our created tree structure using a tree policy. For this, we use the Upper Confidence Bound 1 formula as described in [18]. This formula balances between exploration (select a node in areas that have fewer simulations) and exploitation (select a node in areas that have high evaluation scores). A small random decimal number is added to the calculated UCB value to act as a tiebreaker. The selection algorithm is recursively applied, starting with the root node and ending in a node that is either not yet fully expanded or has a score higher than the threshold. If this is the case,

we find a suitable level and do not expand the node. After the node is selected, we expand it by applying a not-yet-performed, uniform random move to the puzzle level stored in the node. For each not fully expanded node in the tree, we keep track of which moves have not yet been tried. The puzzle created by applying this move is added to the tree as a leaf of the current node. We then apply a simulation on the puzzle level of the newly created leaf. On a copy of the level, we apply a set number of uniform random moves with which we reach a certain 'depth'. For each move, we look if we can exit the simulation early by calculating the score for the current level and checking if it reaches the threshold, if it does, we exit the simulation early. After applying a number of random moves, the score is calculated for the final puzzle level resulting from the simulation. Using backpropagation, this found score is backpropagated in the tree, recursively updating the parents' scores by adding the simulation score to it. It also updates the number of simulation runs in each of the parent nodes. We stop when the root node is reached.
Once all iterations are complete, we identify the best move to apply to the root level. This is done by examining the root's direct children and selecting the child node with the highest visit count. If multiple nodes are tied, we choose the one that was added first, as the order of child nodes is random.

3.2 Our 2D Level Description

We take a level definition as the search field of our MCTS algorithm. A general level is defined by a starting point, an objective, and the space in which the objective must be completed according to [19]. In this thesis, we define a level as a graph where the nodes represent rooms and the edges represent doors between rooms. Each room can contain challenges, collectibles and other game elements. In our case, rooms have a number of colored buttons which open doors of the same color when pressed. Some doors do not have a requirement and are always open. At the start of the level, all colored doors in the level are locked. Doors connect exactly two different rooms and there may be multiple doors between two rooms. After pressing a button, only one type of button color can be active at a time, which adds to the puzzle element of our game. The level begins in one of the nodes of the graph with no buttons pressed, for simplicity we call this Room 1. We place the level exit in Room 1. This exit has a requirement: a specific button color has to be pressed in order to open it. The goal for the player is to find and press the correct button within the level space and make their way back to Room 1 to complete the level. The level must have at least one room.
These rules can be applied broadly, as a graph can be structured in many ways. We restrict the graph to make it better fit our desired final result. One of these restrictions is that the graph must be planar, meaning that there exists a graph embedding in which there are no overlapping edges. This requirement is needed for it to be possible to place doors between connected rooms on a 2D plane. Another restriction is that all rooms must be reachable from the starting room. We can reduce the graph by removing any disconnected room not reachable from the starting room. Finally, there must be at least one solution and not all rooms need to necessarily be visited to find a solution. Figure 2 shows an example of a level according to our description.

3.3 Scoring Criteria

In addition to the requirements specified in the level description, we define specific game-related criteria to make puzzle levels more challenging and to identify incomplete levels, which we refer to

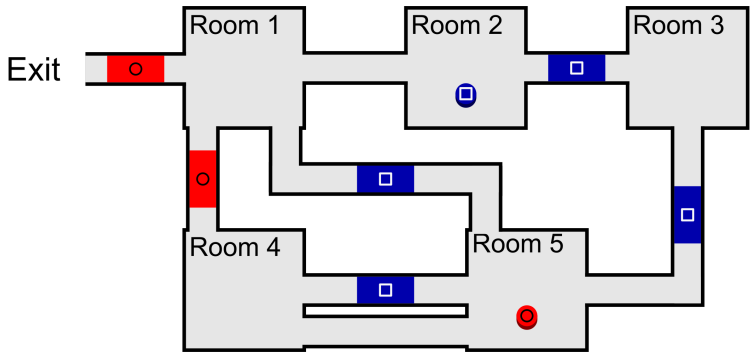

Figure 2: This is a visualization of the level description. There are 5 rooms and 2 button colors (red and blue). The player starts in Room 1 with no buttons pressed, and has to press the red button to escape the level. There are 2 solutions for this level (1-2-3-5-4-1 and 1-2-1-5-4-1).

as defects. We use a scoring system in which the total number of points earned results in a decimal value between 0 and 1. This system is divided into three categories: (1) basic structural criteria, (2) "must-haves" that ensure adherence to the level description, and (3) computationally expensive checks for more challenging levels. Each successful criterion contributes to the final score, weighted according to its importance.

3.3.1 The Basic Level Structure

The criteria required for the basic structure of a level are harsh, as we want to limit the computationally heavy parts later on. If any of the criteria is not strictly met, it is better to fail fast: the final score is calculated and we do not score the level any further. The first criterion is that the button required to solve the level must be placed somewhere in the level; if there are two or more rooms, it cannot be placed in Room 1, as the level could be solved without exploration. Another criterion is that we check the number of total connections. Too few connections means that not all rooms can be connected, and the puzzle becomes trivial if there are too many connections. Similarly, we check the total number of buttons placed in the level and if all specified colors appear at least once in the level.

3.3.2 The "Must Haves"

For the level to pass the "must haves" criteria part, we check for two specific requirements which are necessary to complete the level. The first one is that all rooms are indirectly connected to the starting room. This helps filter out levels for the expensive criterion of checking if all rooms are reachable from the starting room by pressing any number of buttons. The other criterion checks if there is at least one solution to the level, meaning it is playable. If both criteria are met, we continue to the computational heavy part. Otherwise we return the current score.

3.3.3 Expensive Criteria for a Challenging Level

In the final stage, we fine-tune the score based on our ideal level structure. The first criterion is the number of unique solutions. The level would be easy to solve if this number is too high, which is why we restrict it to an (arbitrary) maximum of 5 solutions. Conversely, we give a small penalty if there is only one solution, as we require the level to have multiple solution paths.
We then evaluate each solution path based on three criteria: whether it passes through all rooms in the level, whether it meets a minimum path length, and whether it involves all the different button colors. We add a score based on how many of these criteria are (partially) met.
Finally, we check the planarity of the entire level graph using the linear-time algorithm as explained in [20]. This ensures that the layout can be represented in two dimensions. This is a requirement for the final transformation into the *Rescue Elite* level layout, as detailed in Section 4.1.

3.3.4 All Possible Solutions

One of the grading criteria involves checking the number of unique solutions a level offers. For large graphs with many nodes and edges, this becomes computationally expensive. To address this, we use a modified version of Breadth First Search (BFS) to find all valid paths starting from Room 1, with no buttons pressed, and ending in Room 1 with the required button activated.
BFS relies on marking already visited states to avoid redundant computations. We define a state as a combination of the current room and the currently pressed button. From each state, two transitions are possible: (1) moving to a directly connected room, provided the connection's button requirement (if any) is satisfied, or (2) pressing a button available in the current room, which updates the button part of the state. We stop expanding a path under three conditions: if it revisits a previously visited state, if there are no further valid actions, or if it reaches Room 1 with the correct button pressed.
In the process, we collect all valid solution paths. To enforce our design criterion (limiting the number of solutions to at most five), we terminate the BFS early if more than a set number of solutions are found. Algorithms 1 and 2 outline this process.

Algorithm 1 Compute Number of Unique Solutions.

Input: integer `stop_after_n_finds`
Output: All unique solution paths.
begin
    visited ← empty set
    paths ← empty list
    current_path ← empty list
    `DFS`(0, stop_after_n_finds, visited, paths, current_path)
    return paths

Algorithm 2 DFS Function to Recursively Find Solutions.

Input: integer `state`, integer `stop_after_n_finds`, set `visited`, list `paths`, list `current_path`
Output: Adds all solution paths from `state` to `paths`.

```
begin
    if state is already visited then
        return
    if number of paths ≥ stop_after then
        return
    Add state to visited
    Append state to current_path
    if state is in Room 1 and state has the required button active then
        total_solutions ← total_solutions + 1  Add current_path to paths
    else
        foreach move in the list of possible moves from state do
            Helper(move, stop_after_n_finds, total_solutions, visited, paths, current_path)
    Remove state from visited.
    return
```

3.4 Grading the Outputs

The output of the MCTS algorithm is graded with a value between 0 and 100. The higher the grade, the better a level description is. We grade the level based on four different categories. First, we check the basic requirements of a level. In the second category, we grade the playability of the level by checking if there is a solution, and if every room is reachable from Room 1. The third category is to check the quality of the solution paths. Finally, we have three different AI agents which try to solve the level. Based on how they perform, the grade is increased. In Table 1, the components of the final grade are described. Note that we do not have any early exits like the scoring criteria from Section 3.3 do.

Points may be given partially for some of the criteria; for the others, points are given if and only if the criterion is met. Take, for example, the criterion "Right number of buttons per room". If this criterion is partially met, we reduce the points given by $\frac{5.0}{\text{total number of rooms}}$ for each room that does not meet this requirement. The points given are no bigger than the points mentioned and are not lower than 0.

The three AI agents are used to evaluate different play styles and solution paths. The suboptimal agent searches for a solution path and assesses whether it is sufficiently challenging by evaluating its length. The greedy agent attempts to solve the level using a greedy strategy and a limited number of moves. This may cause it to get stuck in certain parts of the level. This agent is used to detect whether the puzzle can be solved using simplistic logic. Finally, the limited-information agent simulates a player with incomplete knowledge and evaluates whether the level can be solved without relying on trial-and-error. Its goal is to penalize levels that require unfair guessing or obscure logic, which could prevent players from solving them reliably.

Category	Criterion	Points
Basic	Every button color is used.	5
	There are not too many buttons in the level.	5
	The required button color is in the level.	2
	If we have 2 or more rooms, the required button color is not in the starting room.	3
	Every room has at least one connection.	5
	The level has at least one unlocked connection.	5
	The right number of total connections.	5
Medium	The right number of unique button colors.	5
	The right number of buttons per room.	5
	All rooms can be reached from Room 1.	5
	All connections can be reached from Room 1.	5
	The level has a solution.	10
Advanced	The solution count is between 2 and 5 (inclusive).	5
	The length of each solution is long enough.	5
	All unique button colors are used per solution.	2
	All placed buttons are used over all solutions.	4
	All rooms are used over all solutions.	4
AI Agents	The suboptimal agent can solve the level.	10
	The greedy agent can solve the level.	5
	The limited-information agent can solve the level.	5
Total	Basic (30) + Medium (30) + Advanced (20) + AI Agents (20) =	**100**

Table 1: This table shows the level grading evaluation criteria.

3.4.1 Suboptimal Agent

The suboptimal agent contributes a total of 10 points to the final grade. These points are only given if and only if the agent manages to find a solution to the level by making a total number of moves between $[n + b, (n + b) \times 4]$. This ensures that the solution is not too simple and not too difficult. If the found solution falls outside of these bounds or when no solution is found, no points are given. In the strategy of the suboptimal agent, we look at the most optimal move from the agents current position, starting from Room 1 with no buttons pressed. We calculate this move by using the shortest path found by the algorithm from Section 3.3.4 to the current state. The agent takes the optimal move with a 80% probability. In the other case (0.2 probability), the agent makes a random move: Out of the list of possible moves, the agent takes one move with a random, uniform probability. An example of the agent's path is given in Figure 3, where it is shown how the agent's path is not always optimal. The agent continues until it completes the level or reaches the limited number of moves it can make.

3.4.2 Greedy Agent

The greedy agent makes moves according to a greedy strategy and tries to find a solution to the level within a set number of moves. Here, the number of moves is between the range $[n \times 1.2, n \times 5]$,

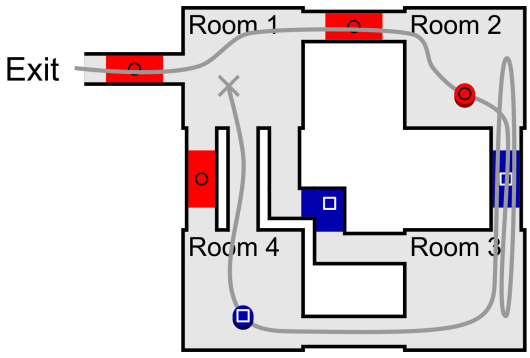

Figure 3: A level representation with a gray path walked by the suboptimal agent. It starts at Room 1 and tries to reach the exit by pressing the red button. The suboptimal part is shown between Rooms 2 and 3, where the agent loops back from Room 2 into Room 3. This event is randomly triggered with a probability of 20%.

and the 5 points are given if and only if the agent finds a solution within this range. In the strategy, the agent only looks one move ahead, meaning it lacks information about the full structure of the level. It looks at all the possible moves it can make from its current position, assigns a score to each move, and takes the move with the highest score. The points are assigned to a move in the following order:

1. If the move leads to a winning state (that is, being in Room 1 with the required button color pressed), the move is awarded 100 points as it completes the level in the next state.
2. If the required button color is in the current room and the agent does not yet have such a button pressed, the move scores 20 points, as pressing this button brings the agent closer to the goal.
3. If there is a button color required to complete the level in a neighboring room, and the agent does not yet have this button pressed, the move is awarded 8 points.
4. All other moves are worth 3 points.

After the score is calculated, we give penalties for the edge cases. Each move gets a large penalty of -22 points if the move ends up in a state where the agent is stuck, meaning no more moves can be performed. For example, when pressing a button which leads to the open doors being closed and the agent cannot go through any of the other doors of the room. Another smaller penalty of -5 points is given for when the next move leads back into a state from which the agent just came (e.g., **State A** $\rightarrow$ **State B** and **State B** $\rightarrow$ **State A**). The last penalty of -2 points is given when the agent activates another button color when the required color is already pressed. Finally, we add a very small random decimal to the moves to manage any tiebreakers.
The greedy agent performs well on easy levels, but it becomes difficult when the level requires the player to look multiple steps ahead, as shown in Figure 4. The greedy agent continues until it completes the level or reaches the limited number of moves it can make.

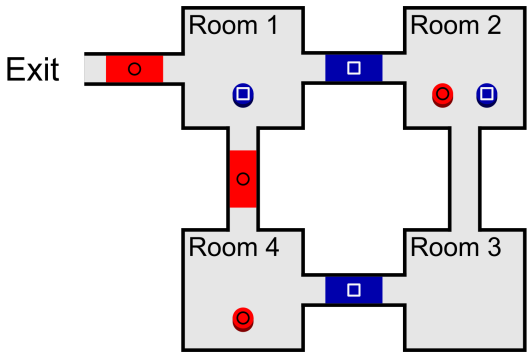

Figure 4: A representation of a level on which the greedy agent fails. In Room 2, the agent presses the red button, as that is a high scoring move, but this locks the agent in Rooms 2 and 3. The agent cannot get out of these rooms (e.g., by pressing the blue button in Room 2 and completing the level by pressing the red button in Room 4), because it prioritizes pressing the button to exit the level (red in this case).

3.4.3 Limited-Information Agent

This agent plays the level similar to how a player would play the level. It uses an optimal strategy, but has limited information about the level. It explores the level and, using the information of the previously visited rooms, it tries to make an optimal move. However, in the case that the agent has to make a decision between multiple actions that lead to unknown outcomes, it always takes the least optimal move, as illustrated in Figure 5. The least optimal move is found by checking whether the agent gets stuck after executing said move. The agent scores 5 points if it finds the exit, but no points are awarded if it gets stuck in the level. There is no restriction on the number of moves it makes.

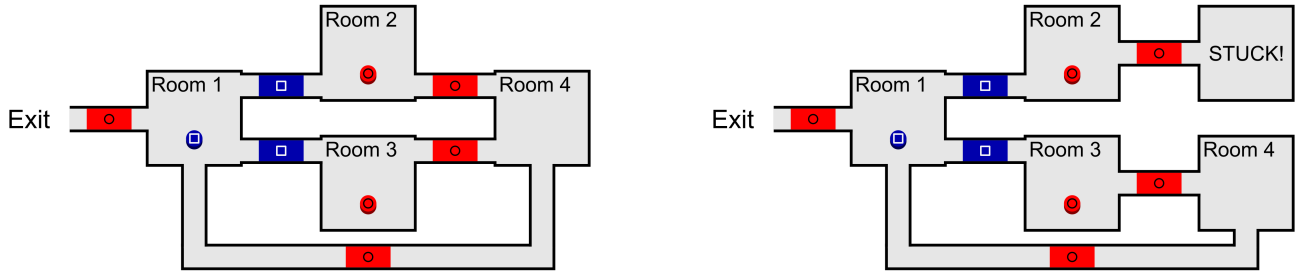

Figure 5: The Limited-Information Agent can solve the level on the left as both choices lead to the same outcome, but it cannot solve the level on the right as it always takes the least optimal choice when a decision has to be made. In this case, that results in pressing the red button in Room 2, which leads to a dead end.

3.4.4 Defects

We describe a defect as a level description that has an insufficient grade. To identify such a level, we consider the first 80 points from the grading, which can be obtained by checking all the structure boxes. If a level scores all 80 points (so 100%), it is a good level, all other levels that score below 80 points are called defects, as they have something lacking. The grading scores given by the AI agents are disregarded when finding defects.

3.5 Uniqueness of a Level Description

The output of Monte Carlo Tree Search is a level description, and we can calculate a grade for each of these levels and determine if they are good or not. However, we cannot say anything about the uniqueness of the levels that are found. For example, all the levels could come down to the same level structure but different assignments of names for different rooms and buttons, meaning our MCTS algorithm finds nothing of significance. To distinguish levels, we define a method to differentiate levels based on their solution paths. Using this method we can better test the performance of MCTS. Only good levels are considered when talking about the uniqueness of a level.
To calculate the uniqueness encoding of a level, we first need to find all the possible solution paths using the algorithm from Section 3.3.4. Each solution path is encoded into a bit string which consists of 1s and 0s, has the length of the solution path, and is read from left to right. In this bit string, a 1 stands for traveling between rooms (e.g., **Room A** $\rightarrow$ **Room B**) and a 0 encodes pressing a button. This encoding removes the need of names for different rooms and buttons. Finally, we create a list of bit strings were each bit string stands for a possible solution of the level. A good level has at least 2 bit strings and can have up to 5. The list is ordered by the length of the bit strings. There are no duplicate bit strings as all solutions are unique.
To compare if two levels are similar (meaning not unique), we compare the length of the list of bit strings. If this length is equal, we compare the bit strings with the same index in the two lists. If all of these bit string pairs are identical (meaning they have the same encoding), we call both levels similar. Otherwise, the levels are unique compared to each other.
We make use of a brute force method to find all the unique levels given the settings of a level description, being the number of rooms and the number of unique buttons. We go over all the $2^{\text{number of moves}}$ possible level combinations, and check if the level is good. We do this by first applying the 'cheap' scoring criteria and checking if this score is above a certain generously low threshold. If this is the case, we check whether the level is good by using the modified, expensive grading function to see if the score is 80 out of 80. For each of the good levels, we apply the uniqueness encoding. For each newly found good level, we compare its encoding to that of all the already found unique encodings. We add the new encoding to the list of uniques if it is not similar to any of the already found unique encodings. After going over all the level combinations, we have obtained the number of unique levels, and a list containing the unique level encodings. An example of two unique levels is given in Figure 6, and an example of two similar levels is given in Figure 7.
In our experiment, we brute force the parameter combinations that have a low search space. The search space increases exponentially, making it almost impossible to brute force the unique levels in our experiment which have higher parameter combinations. We could potentially prune the search space by taking a good look at our scoring and grading functions, but the exponential growth makes this significantly difficult and it falls outside the scope of this thesis.

Figure 6: These two level descriptions are unique compared to each other. The level on the left has 5 solutions and the level on the right has 2 solutions.

Figure 7: The two level descriptions given here are similar. Both levels have the same 4 solutions. These solutions are given by the following bit strings: '01011', '01110111', '01101', and '01010101'. One observation about these two levels specifically is that they are diagonally mirrored (from top-left to bottom-right).

4 Level Generation using PCG

In this section, we highlight the applicability of our method in actual game contexts using a playable prototype we call *Rescue Elite*. This game showcases how the graph-based level descriptions produced by our MCTS approach (Section 3) can be seamlessly converted into structured, interactive environments.

4.1 Rescue Elite

The game *Rescue Elite*[1] is a recreation of the Commodore 64 game *Fort Apocalypse*. The goal of the original game is to rescue people in an active warzone while avoiding enemies, maneuvering through cave-like level structures, and reaching the ending platform. Our game adds to this by introducing a puzzle element. Each connection between rooms may have a locked, colored door which can be unlocked by pressing any button of the same color found in the level. However, only one button may be pressed at a time. The new goal is to rescue all the people while also pressing the correct button and escaping the level through the exit door in the starting room. The difficulty

[1] https://github.com/PerrivandenBerg/RescueElite

of the game has been slightly reduced by adding a HP system, meaning the player's helicopter can take multiple hits before crashing and restarting the level. Figure 8 shows the differences between the two games.

(a) Screenshot from our game *Rescue Elite*.

(b) Screenshot from the original *Fort Apocalypse*.

Figure 8: Visual comparison between our prototype and the original inspiration.

4.2 Generating Levels for Rescue Elite

We apply many different methods to generate levels from an adjacency graph description which MCTS generated as an output. In this part we explain the steps of our algorithm with which we create a level. At the end, the generator outputs a playable level in JSON format, which we can import to the game *Rescue Elite*.

4.2.1 Planar Graph Embeddings

The graph produced by the MCTS algorithm represents a good level. One of the key requirements for such a level is that the graph is planar, meaning that there exists an embedding on a 2D plane without any edges crossing. This property is essential to make a functional puzzle game. If edges intersect visually when they should not actually be connected in the game logic, players may misinterpret the layout and could potentially bypass crucial steps, reducing the intended difficulty of the puzzle.

There are alternative approaches for rendering non-planar graphs on a 2D surface. One example is making use of different z-levels to simulate depth and avoid edge overlaps. In this model, edges can visually pass "behind" one another in a 3D space while presenting it to the player as a 2D space. However, while this concept may solve the edge-crossing issue and remove the need for planarity, it introduces new problems. Complex graphs that are illustrated with this method lead to confused players who have difficulties navigating to the level and reaching the goal. For both practical gameplay reasons and for the generality of the problem (as most levels do not make use of this idea), we chose not to implement this approach.

There are several efficient methods for generating planar graph embeddings, as described in [21] and [22]. These algorithms can produce planar embeddings in linear-time. However, the resulting graph

layouts tend to lack the visual and structural qualities we expect from a thoughtfully designed puzzle level. They often appear technical, and fail to evoke the creative structure desirable in a level.
One potential solution is to use these strict embeddings as a base and then apply transformations to produce a more natural or "randomized" appearance while maintaining planarity. However, this method introduces significant complexity. It requires modifying the layout without causing edge intersections, which can quickly negate the benefits of the linear-time complexity. We impose further, complicated constraints on the created planar graph for gameplay purposes. These restrictions relate to the placement of the nodes, as nodes should not be too close to each other but should still be contained within the 1 by 1 plane, and the angle between the connections, as having two connections too close to each other may result in connections merging together in later steps. By following these constraints, we make it easier to generate the level in later stages of the algorithm. To satisfy these constraints, we iteratively generate node positions until a valid configuration is found. If no valid embedding is found within a reasonable timeframe, we progressively relax the constraints. A good example is illustrated in Figure 9.

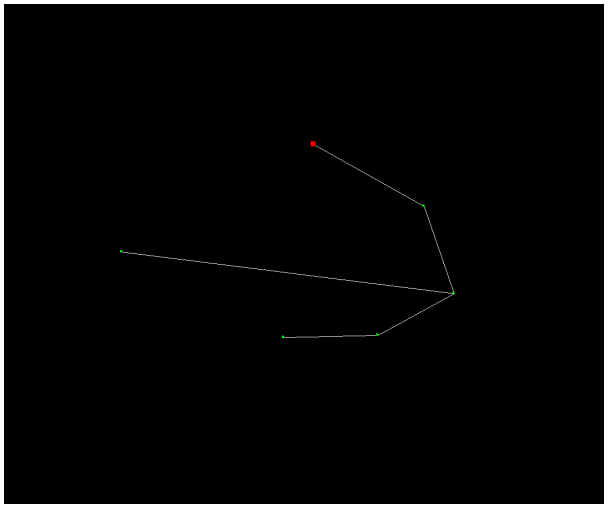

Figure 9: A generated planar embedding using the MCTS-generated level description. Lines represent graph connections, and the green and red nodes indicate node (room) positions. The red square at the topmost point marks the starting room.

4.2.2 Creating Room Structures

After we have the coordinates of each of the rooms, we create a grid that has the size of the final level and contains ID values for each of the different rooms and information about where to place walls. We transform the coordinates from the planar graph embedding to fit in the just created grid. Using the coordinates, we take for each node in the graph the closest cell and mark it with a unique ID. This cell is the 'center' of the room.
After placing all the room cells, we connect the rooms by drawing lines between two connected rooms in the grid. Since the graph is planar, none of these lines overlap. We split each of the lines between two rooms in half; the ID of the first half of the line is set to the ID of the closest room, and vice versa with the other half of the room.

Finally, we make use of a Cellular Automaton to expand the cells belonging to each room ID. We do this over a set number of iterations and for each cell, we copy the ID of the orthogonality and diagonally neighboring cells. In case a cell has multiple neighbors with different room IDs or if it is bordering the edge of the level, we mark that cell as a solid wall and do not change it over the iterations.
The final structure looks like a cave system consisting of multiple distinct rooms. An example is shown in Figure 10 where we have a total of 5 rooms.

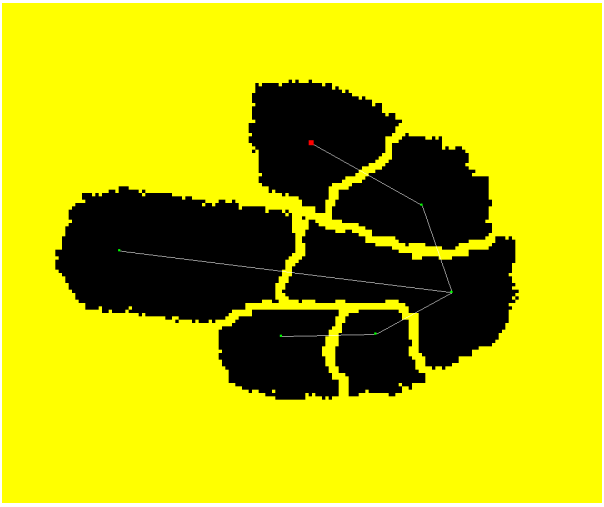

Figure 10: A generated cave-like structure using the generated planar embedding. Here, the yellow pixels represent walls and the black pixels represent the sky in which the helicopter can move around.

4.2.3 Placing Doors Between Rooms

After generating a basic structure for the level, we simplify it by decreasing the size of the grid so that each 4 by 4 grid of cells of the old grid is converted to either a wall cell or a cell with a room ID in the new grid, making the size of the new grid a quarter of the original size. This simplification step makes it easier to place doors between rooms as we are working with fewer cells in the grid.
To detect possible placements for doors, we look for cells that neighbor exactly two different rooms and mark them. Using the connection description from the MCTS output, we randomly add doors between two rooms using the marked cells. Here, we make that doors are not too close to each other. Each placed door is assigned a color according to the level description. If we cannot place the required number of doors, the level generation fails and we restart the generation process.
After having placed all the doors correctly, we add an exit to the level. This exit tunnel is placed in the starting room and has a door as well with the color of the level's completion requirement. If this exit tunnel cannot be placed, for example when the room is in the center of the level and has other rooms surrounding it, we restart the level generation process. The player completes the level by going through the open door.
An example of a succesfully generated level up onto this step can be seen in Figure 11.

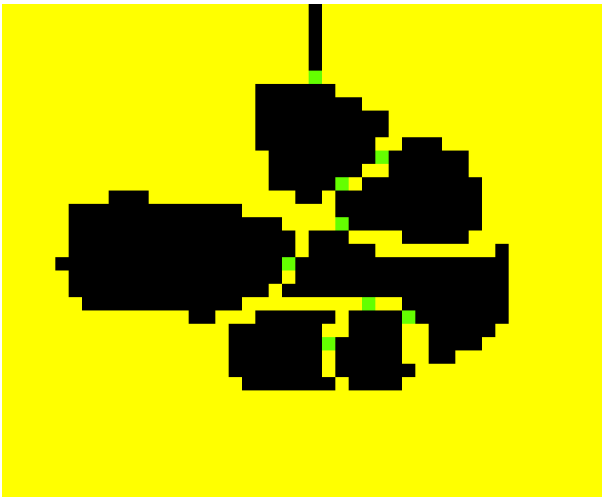

Figure 11: A simplified version of the cave-like structure with the lime cells indicating door locations. Every cell represents 4 by 4 pixels. The long tunnel to the border of the level indicates the exit.

4.2.4 Placing Objects

The game *Rescue Elite* has game elements which need to be placed in a level. These are objects like the player's helicopter, the buttons, the fuel station to restock the helicopter's fuel, the people that need to be rescued, and enemies. To place these objects, we scan the level for possible placement positions on both the ground (e.g., people and enemy tanks) and in the air (e.g., enemy drones and hearts). Since a cell of the minimized grid is 4 by 4 cells in the actual grid, we can use 1 cell for smaller objects, and 2 horizontally neighboring cells for larger platform types (e.g., Starting platform, fuel station, and the buttons).
The player object is strictly placed in Room 1. For the player, we first place down a platform on which the player can land their helicopter, and then we place the helicopter above the platform. Regardless of where the helicopter is placed, it faces to the right. Similarly, the buttons are also strictly placed in the correct rooms according to the level description.
For the other objects, we randomly place them around the map on either the floor (if it is a ground object) or in the sky (if it is a sky object). Only one object can be placed per cell. Ground objects cannot be placed on top of doors, and sky objects cannot be placed too close to the walls. Each level contains a random mix of objects, the number of which scales with the number of rooms.
The final result of this step is illustrated in Figure 12.

4.2.5 Detailing the Terrain

After placing the objects, we can start detailing the terrain. Placing objects and doors on an inconsistent terrain is difficult; that is why we first place all the important parts and, if successful, we make the terrain more rough.
For the roughness of the terrain, we start by increasing the grid to its original size (by multiplying every cell's size by 4). The terrain is now very blocky and not interesting. Using a complex Cellular Automaton, we connect the 4 by 4 grid cells into a very structured level, an example of which is illustrated in Figure 13. The CA used here ignores all the cells containing an object and all the cells directly underneath a ground object. This way the objects are not hindered in any way.

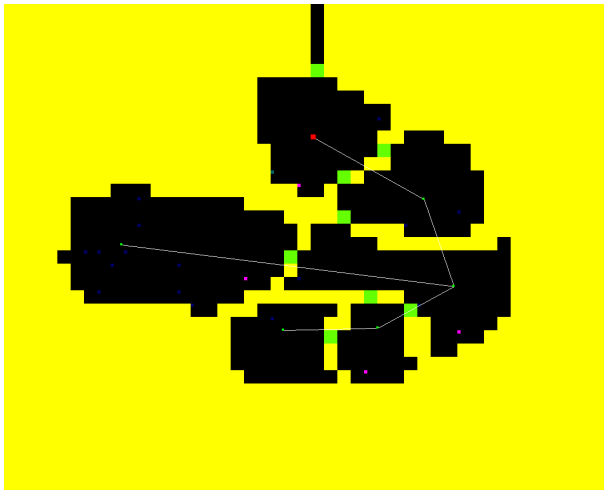

Figure 12: On the simplified version, objects are placed and marked by the left-corner pixels. The magenta pixels are the buttons, the dark blue pixels are any object, and the dark cyan pixel in the starting room is the player's starting position.

Using a randomization algorithm, we add roughness to the terrain. Every wall cell that has a sky cell as a neighbor has a small probability to change into a sky cell as well. By repeating this algorithm for a few iterations, the level looks very rough.
Finally, using a CA we smooth the terrain (both the walls and doors) for a set number of iterations. The final result is shown in Figure 14 and has a more cave-like structure.

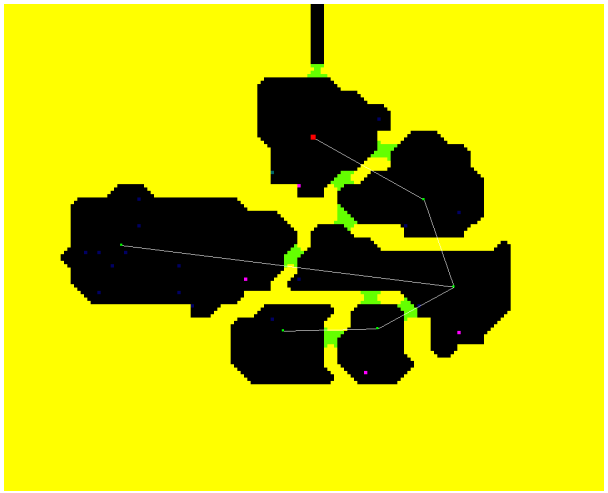

Figure 13: The level after applying the CA which brings structure to the level. Note that the cells containing an object are not changed by the CA.

4.2.6 Exporting to Rescue Elite

After generating the level, we export the grid encoding into a JSON format. This format can be imported into the game *Rescue Elite* and be played as a functional level. Figure 15 shows the level layout in the game when importing the level from Figure 14 as JSON format.

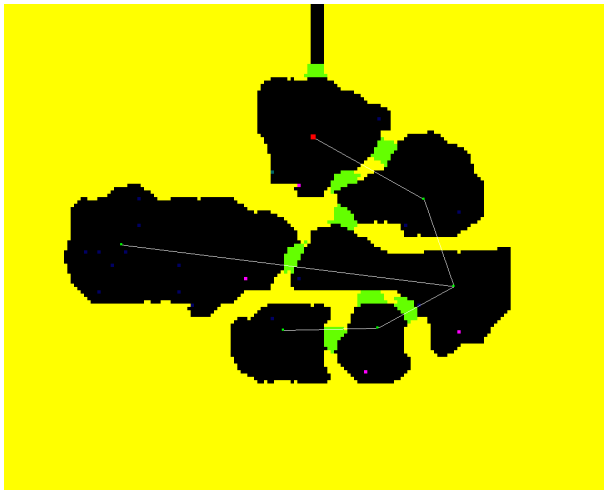

Figure 14: We randomize and smooth the structured level. This results in the final level which can be imported to the game.

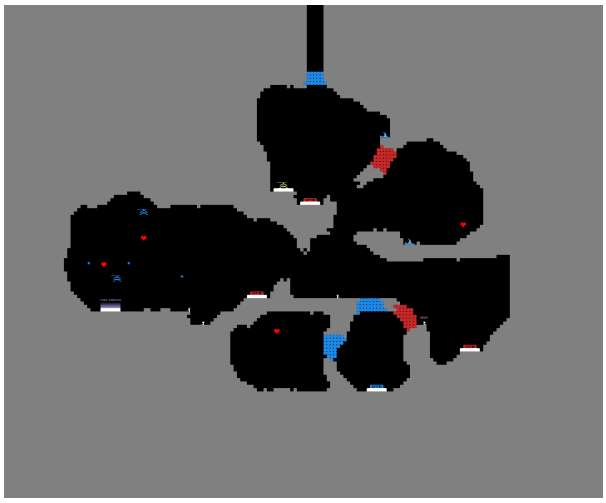

Figure 15: One of the levels generated by our PCG pipeline imported into the game *Rescue Elite*.

5 Experiments

In this section we detail the experiments performed on the output of MCTS, and explain how Rescue Elite was surveyed.

5.1 Monte Carlo Tree Search

We did two different experiments for MCTS. In the first experiment, we examine the number of unique solutions that can be found over all the level combinations given the parameters. We explain how these are obtained and the difficulties of the search space. In the other experiment, we identify the best parameters for MCTS and explain how we measured the performance. The button color required to escape the level stays consistent over the experiments.

5.1.1 Total Unique Levels

In this experiment, we brute force the all the level possibilities given the parameters to find the encodings of good, unique levels. For this experiment, there are two parameters to change: the number of rooms and the number of unique buttons. We range the number of rooms $n \in [2, 3, \ldots]$, and we take the range of unique buttons $k \in [1, \ldots, n]$. As the search space grows exponentially when increasing these parameters, we only consider cases where the size of the search space is lower than 2^{32}.

5.1.2 Performance

To measure the performance of the Monte Carlo Tree Search algorithm, we make use of the previously defined Grading Function. We run MCTS and wait for an output which is given a grade. Over a number of runs, the average of all the grades is calculated. This, together with the number of found unique solutions and the number of defects, gives information about the performance.
There are many parameters that can be changed in this experiment. The main parameters are the number of rooms and the number of unique buttons. These parameters are used to define all the different levels, and we take the number of rooms $n \in \{2, 3, \ldots\}$, and the number of unique buttons $k \in [1, \ldots, n]$. Other than these parameters, there are parameters which can be tweaked to improve the performance of MCTS as described in Table 2.

Parameter	Description
Epsilon	A small value which is used in the Upper Confidence Bound (UCB1) and controls the exploration vs. exploitation of the algorithm.
Score Threshold	Used for early exits within the algorithm (with building up the tree structure) and outside the algorithm (with the number of moves the agent makes). We exit if the score is higher than this value.
Random Moves	The percentage of random moves applied to an empty level before starting MCTS. The total number of random moves is calculated by multiplying this value with the number of possible moves.
Runs	The number of times we repeat the MCTS pipeline. The final grade is averaged over all the runs.
Brain	The maximum number of nodes allowed in the MCTS tree structure (i.e., the memory/capacity of the algorithm).
Depth	The maximum number of random moves made during a single simulation from a given node.
Moves	The total number of moves the MCTS agent is allowed to make while improving the initial level.

Table 2: MCTS hyperparameters used in our level generation pipeline.

5.2 User Experiments

To validate the application of MCTS in level design, we asked a group of people, consisting mostly of university students, who are familiar with playing games, to play the game *Rescue Elite* and fill out a survey afterwards. The goal of the experiment was to measure the flow of the game as described in [23]. We measured the learning curve of the participants by looking at their frustration levels. Low frustration levels indicate that the levels were too boring and did not challenge the player enough, while high frustration levels meant that the game was too difficult, which disturbs the flow of the game.

The levels of the game were generated by our algorithm with the exception of the first tutorial level which was hand crafted in the style of *Fort Apocalypse*. For each level, we briefly checked by looking at the simplified preview of the levels if it looked possible to complete. Without any further post-processing, we put the levels into the game. The first two generated levels had a total of 3 rooms; the number of rooms increased for every level that followed until we reached a level with 7 rooms. In total we have 8 playable levels out of which 7 were generated.

After completing all the levels, the user was prompted to fill out a survey. As the levels increase in difficulty and we do not know if the player's skills can adapt fast enough to these difficulties, we added an option to fill out the survey early when they got stuck on a level. The survey consisted of three categories of questions. First, we asked the participants about their gameplay experience, specifically about how challenging the levels were and how frustrated they were after playing the game. In the second part of the survey we asked the participants about the levels' design. This part asked about whether they noticed if a level had multiple solutions, and we asked for design features they liked and for design features which could be better or were missing. In the last part of the survey, we asked whether they noticed that the levels were AI-generated, and about their interests in future work using this method.

On the front page of the survey, we asked the participants to upload the level data that they could download from the game's completion menu. This game data contained information about the number of levels they completed, and for each level, which paths led to their helicopter crashing and which path led to them completing the level. By providing the game data in the survey, we could link the filled in answers of the survey to the gameplay of the participant.

6 Results

In this section, we discuss the results from the performance evaluation of MCTS, followed by findings from user experiments involving gameplay analyses and survey responses.

6.1 Quality of MCTS

Table 3 summarizes the outcomes of our MCTS experiments. The left side of the table (up to 'Moves') lists the hyperparameters that influence MCTS performance, and the right side (from 'Average Grade' onward) shows the algorithm's effectiveness, including the number of unique good levels discovered and overall quality.
For levels with a low number of possible move combinations (below 2^{16}), the MCTS algorithm successfully discovered all unique good solutions across 1000 or more runs. For configurations with fewer than 2^{64} possible move combinations, we were consistently able to find parameter settings yielding an average grade above 80. Notably, reducing the Maximum Score Threshold improved performance in larger search spaces as it was easier to find levels that are almost good.
We found that when the number of rooms equals the number of button colors, no good level could be produced. This is not merely due to search failure, but rather a fundamental conflict in the current definition of what constitutes a "good level" under these specific parameter settings. We require good levels to both include all button colors and ensure that each button color is used in every valid solution. This is enforced by placing locked doors in the level that require pressing each button color to proceed. However, when the number of unique buttons equals (or exceeds) the number of rooms, all available connections must be used efficiently. At the same time, another criterion requires that at least one door must be unlocked. This introduces a path that bypasses at least one of the colored button, disqualifying it from being good. This reveals a logical inconsistency: the criteria for a "good level" in these configurations are mutually exclusive, rather than simply hard to satisfy. Brute-force analysis confirmed the lack of good levels for $n = b = 2$ and $n = b = 3$. For $n = b = 4$, MCTS failed to find a single good solution over 1000 iterations. For this definitional conflict, we decided not to experiment any further with configurations of $k \geq n$.
As level complexity increased, we adjusted parameters to improve the search's ability to reach optimal configurations. We noticed that by reducing the proportion of random moves, the algorithm's consistency in generating good levels improved. Increasing the 'Brain' (the size of the MCTS tree) and search depth generally correlated with improved level quality. However, beyond a certain scale (notably for $n \geq 5$ and $b \geq 2$), discovering unique good levels becomes more challenging and computationally expensive.

6.2 Player Experience and Level Flow in Rescue Elite

For the game we did several experiments at once by letting participants play the levels and fill out a survey afterwards. The participants mainly consisted of university students who have have experience in playing games. In total there were 21 participants. We analyzed the game data and the survey data, and the results are discussed in this section.
To evaluate the gameplay experience of Rescue Elite, we conducted a combined study involving direct gameplay and a follow-up survey. The 21 participants with prior gaming experience, most of whom are university students, played through a selected set of levels and provided feedback

n	b	Eps.	Max. Thres.	% Rnd. Moves	Runs	Brain	Depth	Moves	Average Grade	Uniq. Found	Total Uniq.	Defects	Combs.
2	1	0.2	0.97	0.3	10000	100	3	10	86.02	1	1	0	2^4
2	2										0		2^7
3	1	0.2	0.97	0.33	10000	100	30	30	90.003	13	13	0	2^9
3	2	0.2	0.995	0.33	1000	500	10	30	94.0012	51	51	14	2^{15}
3	3										0		2^{21}
4	1	0.2	0.995	0.3	1000	1000	10	30	91.4292	23	23	6	2^{16}
4	2	0.2	0.995	0.3	1000	2000	30	30	92.4278	365	938	104	2^{26}
4	3	0.5	0.95	0.3	1000	3000	30	40	84.5233	42	?	956	2^{36}
4	4	0.5	0.9	0.3	1000	2000	30	30	75.364	0	?	1000	2^{46}
5	1	0.2	0.994	0.33	1000	2000	30	30	92.0727	81	85	4	2^{25}
5	2	0.3	0.95	0.3	1000	3000	30	30	90.1422	174	?	781	2^{40}
5	3	0.2	0.95	0.3	1000	3000	30	40	83.9642	29	?	970	2^{55}
6	1	2	0.975	0.2	1000	5000	30	50	89.2477	56	?	832	2^{36}
6	2	2.5	0.975	0.2	1000	6000	40	50	87.6211	259	?	702	2^{57}
7	1	1.8	0.97	0.15	1000	5000	30	50	88.831	48	?	904	2^{49}
8	1	0.5	0.9	0.15	1000	5000	30	30	80.4069	4	?	995	2^{64}

Table 3: Results of MCTS experiments by number of rooms n and unique button colors b. Only levels with total state spaces below $2^{64} \approx 1.84 \times 10^{19}$ were tested.

afterward. We analyzed both the in-game data and survey responses to understand how players interacted with the levels, where they experienced challenges, and how well the AI-generated content supported a coherent and enjoyable flow.

6.2.1 The survey results

The survey feedback indicated that players generally found the game moderately challenging. Most participants rated the difficulty of levels from moderate to difficult, and a noticeable portion reported above-average frustration levels ranging from "somewhat" to "quite" frustrated. This suggests that while the overall difficulty curve may be appropriate, some design elements (e.g., enemy placement or narrow passageways) may introduce friction that feels more punishing than fair. The players' self-assessed performance followed a fairly balanced distribution, suggesting a balanced difficulty curve across the player base.

The level design itself was received with mixed but generally positive impressions. Players highlighted the puzzle-like structure, variety of challenges, and multiple viable paths as enjoyable features. However, the narrow level design in some of the levels was received poorly, especially when combined with enemy pressure, which often disrupted the pacing and flow. Crucially, none of the participants identified the levels as being AI-generated, demonstrating that the design quality of these levels

was perceived as human-made.
Despite some frustrations, most players expressed interest in playing more levels. This validates that our methods are a promising foundation for continued development.

6.2.2 Frustration Analyses

To verify the subjective survey data, we quantified frustration and performance using in-game events as:

$$\text{Frustration} = \text{Deaths} + \text{Damage Taken}$$

$$\text{Performance} = \text{Enemies Destroyed} + \text{People Rescued} - \text{Frustration}$$

Figures 16,17, and 18 visualize these metrics. Notably, Levels 2 and 6 stood out as sources of significant frustration, a trend observable across all three figures as well as later data (such as death heatmaps in Figure 20). These levels showed high frustration scores and consistent spikes across individual performance curves. This implies that their difficulty exceeded what the average player had learned up until that point in the progression.
Figure 17 shows Levels 2 and 6 appearing visibly darker across multiple participants, reflecting consistent frustration. Furthermore, Figure 18 shows individual frustration curves, with Levels 2 and 6 clearly disrupting an otherwise steady learning curve.

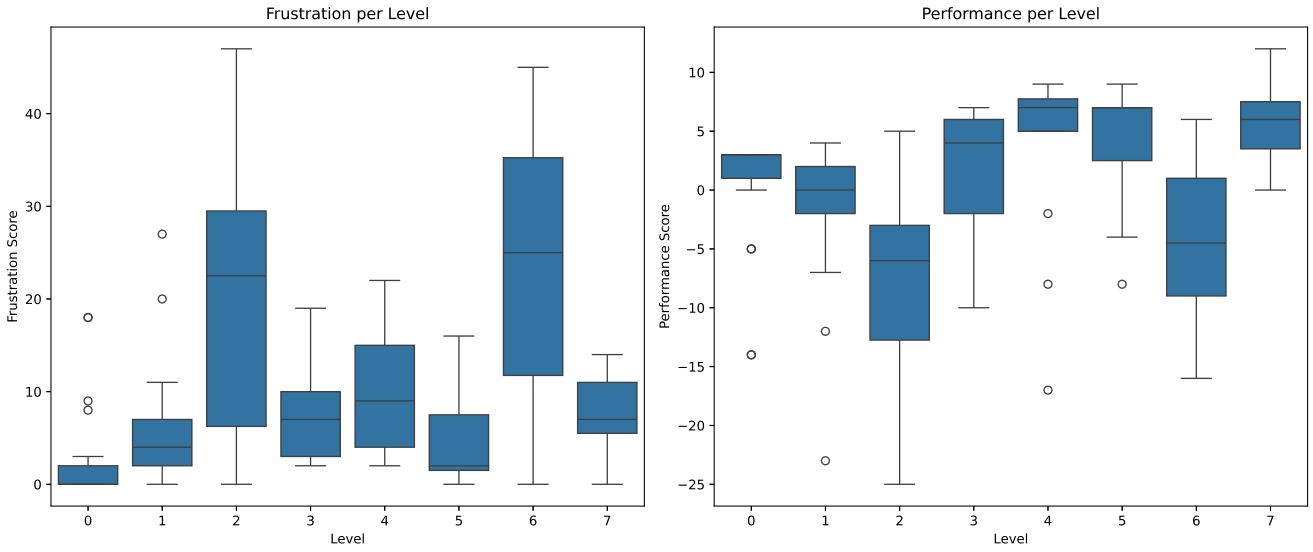

Figure 16: The boxplot on the left shows the frustration score per level with notable spikes in Levels 2 and 6, which disrupt the learning curve. On the right the boxplot shows the players' performance over the levels with again notable spikes in Levels 2 and 6. Outliers in the performance might be influenced by the number of enemies destroyed, as this task does not visually reward the player.

6.2.3 Heatmap of Level Solutions

To evaluate whether the AI-generated levels effectively support multiple solution paths, we analyzed players' paths in the levels through the use of heatmaps. These visualizations show only the successful paths starting from the level's start (dark purple) to the completion of the level (yellow).

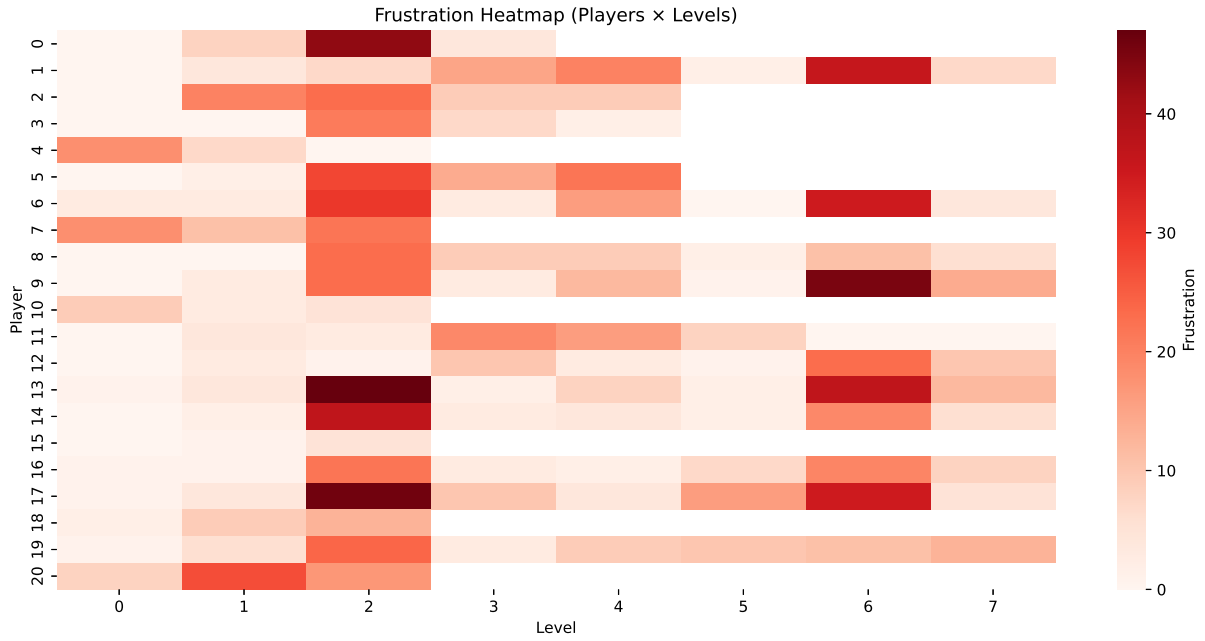

Figure 17: The heatmap shows the frustration over the levels for each individual in the experiment. The frustration goes from not frustrated (light red) to very frustrated (dark red). Some of the individuals left the experiment early and did not complete all the levels. Notable is that the frustration levels of individuals vary in the Levels 2 and 6.

Across most levels, players took diverse paths to complete objectives, indicating that the levels indeed support non-linear, flexible problem-solving. This was one of the intended design outcomes of our MCTS-based level generation. Taking the examples shown in Figure 19, in Level 3, participants discovered two ways to solve the level using the two blue doors in the center of the map; and in Level 4, both red buttons are used, while pressing the top one seems more logical. This suggests that the level design not only allows but encourages exploration and accommodates for different play styles. The solution path heatmaps for all the levels of the experiment can be found in Section A.2.

6.2.4 Frustration Points in a Level

To pinpoint frustration points within the different levels, we created heatmaps based on player deaths and damage events. These frustration maps visualize the paths players took before failing and mark the spots where the player took damage (yellow circles) and where the player crashed (red crosses). The paths are from the start (dark purple) until the point they crashed or restarted the level (yellow). By analyzing these patterns, we can identify where level design may unintentionally cause spikes in difficulty and player frustration.
Levels 2 and 6, as illustrated in Figure 20, once again stand out compared to other levels like Level 1 shown in Figure 21. In Level 2, many players crashed near a particularly narrow passage required to complete the level. In Level 6, a similar bottleneck occurs where an enemy tank obstructs a narrow doorway, leading to repeated player deaths.
In general, the most common causes of player failure across levels where enemy projectiles and tight doors. These insights provide valuable direction for future adjustments and verify the information

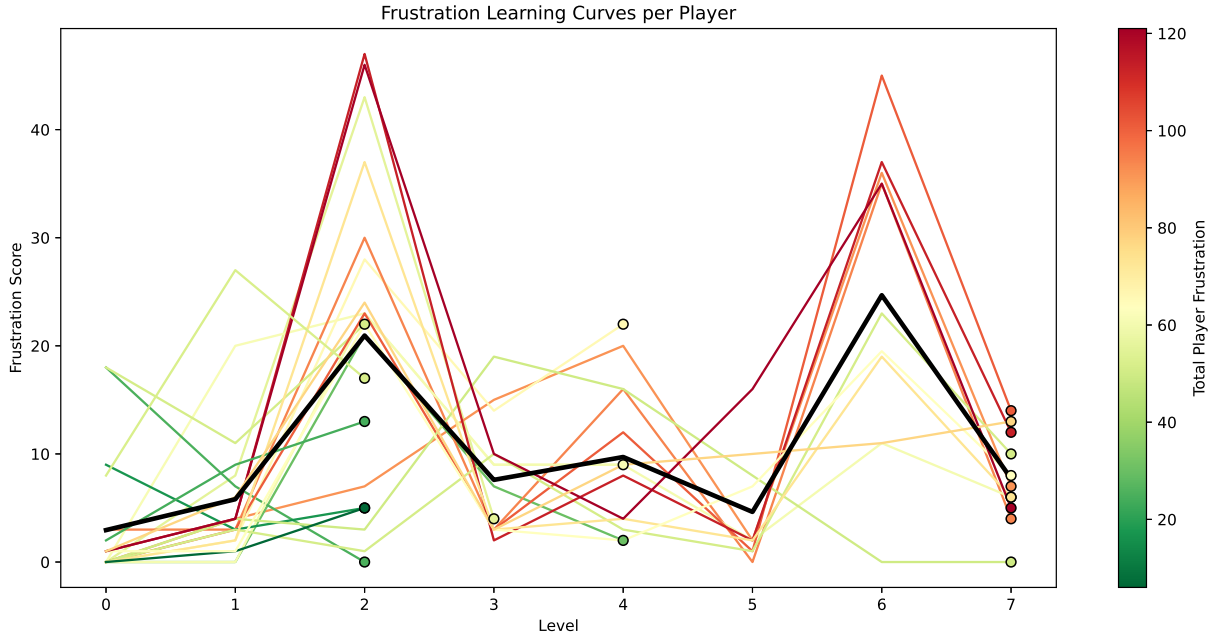

Figure 18: The line graph plots the individual learning curve. Green lines are of participants who were not frustrated and might have experienced the game as too easy, and dark red lines are of individuals who were very frustrated. The black line indicates the average over all participants. Again, Levels 2 and 6 are clear frustration points while we can also see some mild frustration around Level 4. We can also see that Level 5 might have been too easy as the frustration levels were similar to the tutorial level (level 0) for most players.

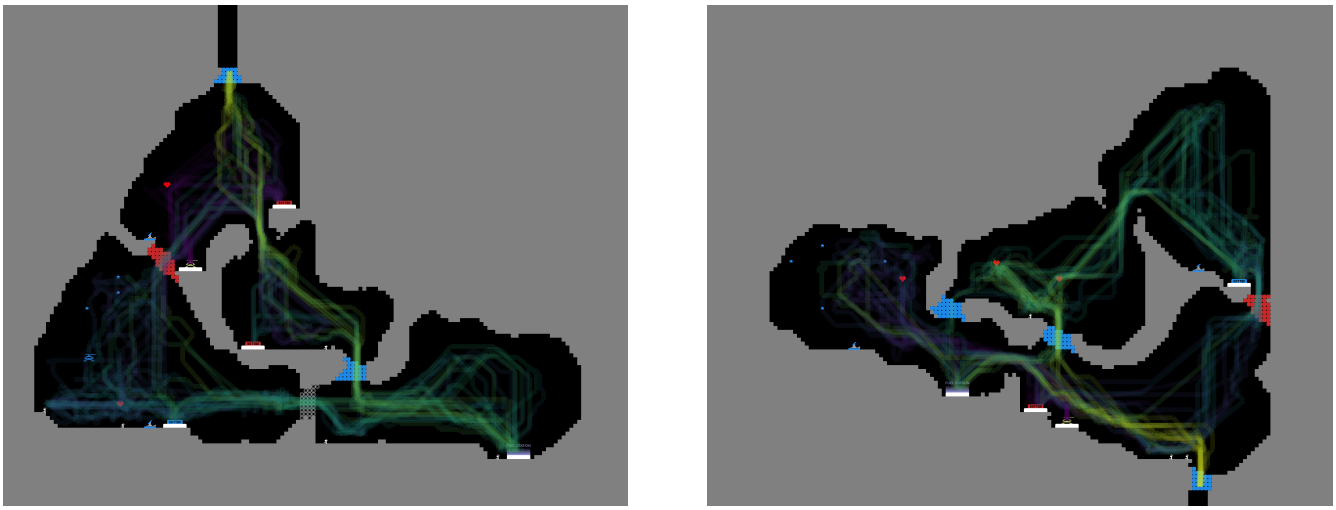

Figure 19: In level 4 (on the left) there are two red buttons. Even though the top red button is easier to press, some participants first explored before pressing a button and ended up pressing the other, lower red button. The layout of Level 3 (on the right) in the game. Players found multiple ways to solve this level, although not all solutions were evenly popular.

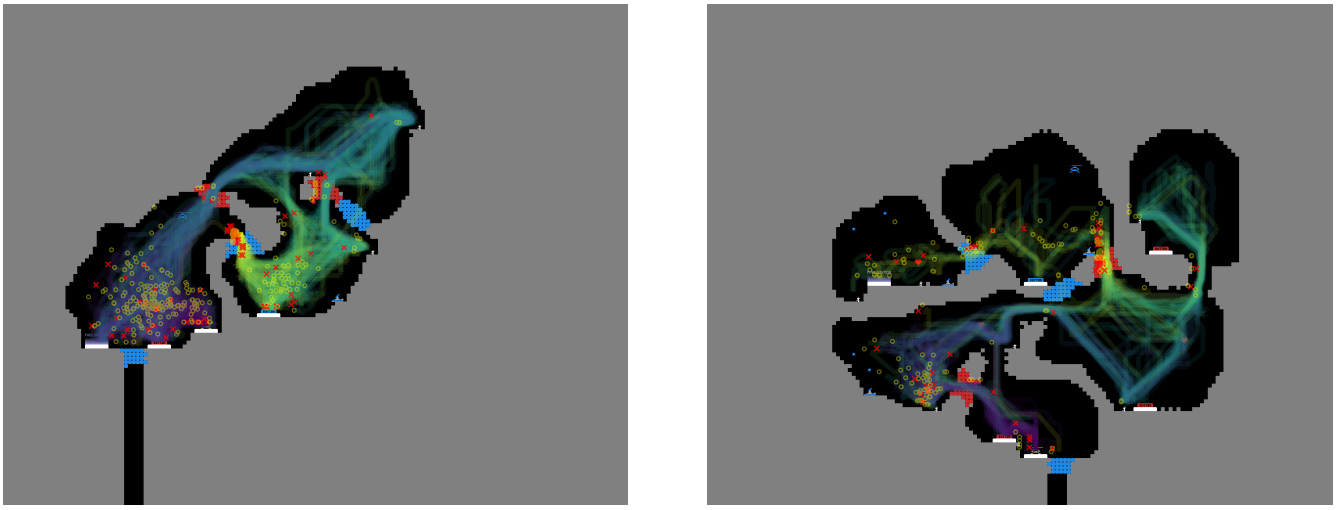

Figure 20: The frustration heatmap of Level 2 is shown on the left. A high concentration of crashes occurs at the blue door in the center of the level, which players must pass through to progress. In the starting area, an enemy helicopter ambushes the player, leading to a significant number of participants taking damage early on. The right figure is the frustration heatmap of Level 6. A dense cluster of failures appears near a bottleneck where an enemy tank blocks the path, causing repeated crashes and preventing many players from advancing.

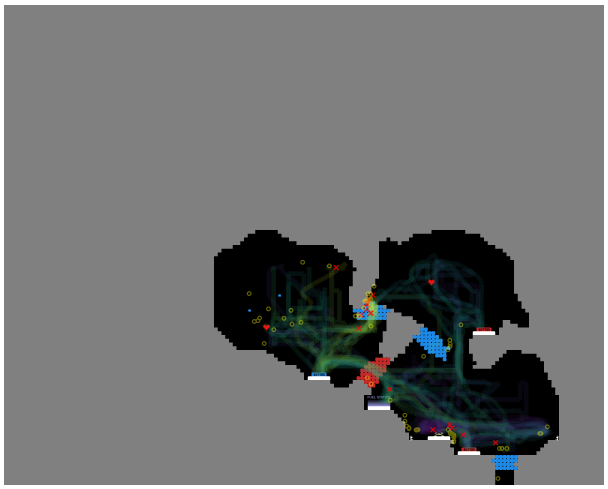

Figure 21: The frustration heatmap for Level 1. Here, the failure points are more scattered around the level that suggests fewer design issues compared to Levels 2 and 6.

from the survey. The frustration heatmaps for all the levels of the experiment can be found in Section A.3.

7 Conclusion

In this work, we explored the use of Monte Carlo Tree Search (MCTS) for procedural level generation in the context of our 2D game *Rescue Elite*. Motivated by the challenge of generating levels that feel human-designed, our goal was to produce content that is not only solvable and varied, but also structurally coherent and supports multiple solution paths. To this end, we proposed a pipeline that integrates MCTS with domain-specific constraints to generate abstract level descriptions in the form of planar graphs. These descriptions were then converted into playable levels using Procedural Content Generation (PCG) techniques tailored to our game's mechanics. Finally, we evaluated the AI-generated levels through gameplay data and player feedback.

Our experimental results demonstrate several promising outcomes. The MCTS-based method proved effective at generating level graphs with meaningful structure and multiple viable paths. Quantitative metrics, including average level grade, uniqueness across generations, and the proportion of imperfect levels, demonstrated the robustness of the generation system across different level configurations. However, we observed a scalability limitation: as the number of rooms increases, the search space expands exponentially. To maintain computational feasibility, level generation was therefore capped at a maximum of 8 rooms.

A user study with 21 participants experienced in gaming showed that the generated levels were generally well-received. Players found the levels moderately challenging, with frustration peaking at specific design bottlenecks. Analysis of player paths confirmed these bottlenecks and revealed that, in levels with multiple solutions, most players discovered at least one alternative path. However, the results also highlighted some notable shortcomings. Levels 2 and 6, in particular, were frequent sources of frustration, mainly due to narrow passages and poorly placed enemies. These issues point to shortcomings in the adaptation of the method, as the conversion from planar graph to playable layout is non-trivial, rather than flaws in the MCTS process itself. While our frustration and performance metrics provided useful insights, they may not fully capture the complexity of player experience.

Overall, our findings show that MCTS, when properly constrained and fine-tuned, provides a powerful and flexible approach to procedural level generation. It achieves a balance between structure, variation, and replayability. These qualities that are often difficult to realize through traditional PCG methods alone. While there is room for improvement, our system demonstrates the potential of MCTS-based approaches in generating human-like puzzle levels with multiple solutions.

7.1 Future Work.

There are several promising directions for future research. The MCTS-based approach could be adapted for other puzzle-like games with complex constraints and solution strategies. Our pipeline could also benefit from being more closely tailored to specific game genres or mechanics, allowing for deeper optimization of design parameters.

One potential extension is the incorporation of dynamic difficulty adjustment (DDA), where level generation adapts in real time to a player's performance or frustration levels. This concept has been successfully explored in PCG contexts, such as in [24] and [25]. Integrating DDA with our MCTS method could result in levels that better align with a player's evolving skill.

Another promising avenue is to explore neural networks as part of the MCTS evaluation phase. Like hybrid methods combining MCTS with deep learning, such as the well-known AlphaGo

architecture [26].
Finally, improved modeling of player behavior and preferences could lead to more personalized and satisfying gameplay experiences.
Together, these directions offer a path toward more intelligent, adaptive, and human-like procedural generation systems.

References

[1] Nathan Brewer. Going rogue: A brief history of the computerized dungeon crawl. *IEEE-USA InSight*, 2016.

[2] Eetu Tirkkonen. Randomness in roguelike games. Master's thesis, Tampere University of Applied Sciences, November 2024.

[3] Sébastien Bénard. The level design of dead cells. *Deepnight Games*, February 2020.

[4] Richard C. Moss. Ascii art + permadeath: The history of roguelike games. *Ars Technica*, March 2020.

[5] Cameron Browne. Towards mcts for creative domains. In *Proceedings of the Second International Conference on Computational Creativity*, April 2011.

[6] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[7] Corinna Jaschek, Tom Beckmann, Jaime A. Garcia, and William L. Raffe. Mysterious murder - mcts-driven murder mystery generation. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[8] Constantine Caramanis, Sarvesh Nagarajan, and Matthew Graves. Procedural content generation of angry birds levels using monte carlo tree search approved by supervising committee:. 2016.

[9] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Fausto Mourato, Manuel Próspero dos Santos, and Fernando Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, ACE '11, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Mark J. Nelson Noor Shaker, Julian Togelius. *Procedural Content Generation in Games*. Springer, 2016.

[12] Y. P. A. Macedo. An integrated planning and cellular automata based procedural game level generator. 2018.

[13] Barbara De Kegel and Mads Haahr. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, 12(1):21–40, 2020.

[14] Norbert Heijne and Sander Bakkes. Procedural zelda: a pcg environment for player experience research. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Joris Dormans and Sander Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):216–228, 2011.

[16] Seçkin Yılmaz and Vasif V. Nabiyev. Comprehensive survey of the solving puzzle problems. *Computer Science Review*, 50:100586, 2023.

[17] Seth Cooper and Mahsa Bazzaz. Literally unplayable: On constraint-based generation of uncompletable levels. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, FDG '24, New York, NY, USA, 2024. Association for Computing Machinery.

[18] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. In *Machine Learning*, volume 47, pages 235–256, 2002.

[19] Jesse Schell. *The Art of Game Design: A Book of Lenses*. CRC Press, Pittsburgh, Pennsylvania, USA, 2014.

[20] John Boyer and Wendy Myrvold. On the cutting edge: Simplified o(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, Jan. 2004.

[21] Lucie Martinet. Drawing planar graphs. 2010.

[22] M. Chrobak and T.H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[23] Arttu Perttula, Kristian Kiili, Antero Lindstedt, and Pauliina Tuomi. Flow experience in game based learning – a systematic literature review. *International Journal of Serious Games*, 4, 03 2017.

[24] Simon Demediuk, Marco Tamassia, William L. Raffe, Fabio Zambetta, Xiaodong Li, and Florian Mueller. Monte carlo tree search based algorithms for dynamic difficulty adjustment. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 53–59, 2017.

[25] Mohammad Zohaib. Dynamic difficulty adjustment (dda) in computer games: A review. *Advances in Human-Computer Interaction*, 2018(1):5681652, 2018.

[26] Huang A. Maddison C. et al. Silver, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(1):484–489, 2016.

A Procedurally Generated Levels

In this section we show images of the generated levels and the results of the user experiment on top of the images.

A.1 Generated Levels

Table 4 shows the levels generated by our PCG method as described in Section 4. The first level is the tutorial level which was not generated.

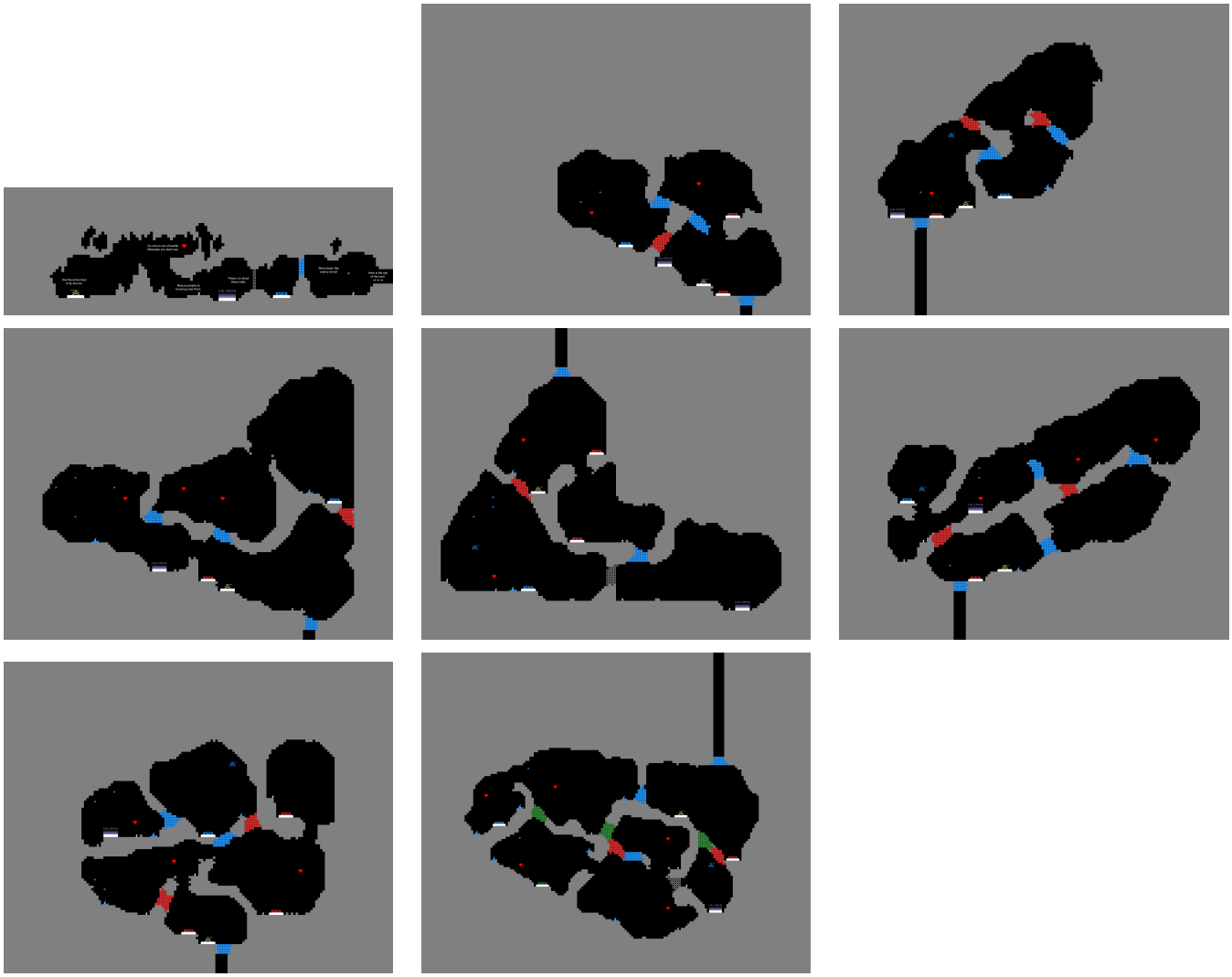

Table 4: All levels generated by our PCG method for the game Rescue Elite. Gray areas represent solid walls, while black areas indicate open space where the player's helicopter can fly freely. Blue, red, and green walls are color-coded doors that require an activated button of the corresponding color to open. Red hearts represent health pickups. The player completes a level by entering the long tunnel that leads out of the level.

A.2 Heatmap of Found Solutions

Table 5 shows the solutions paths found by the players of the game put on top of the generated levels generated. Only the solution path is shown. The solution paths go from dark purple (start of the level) to yellow (end of the level).

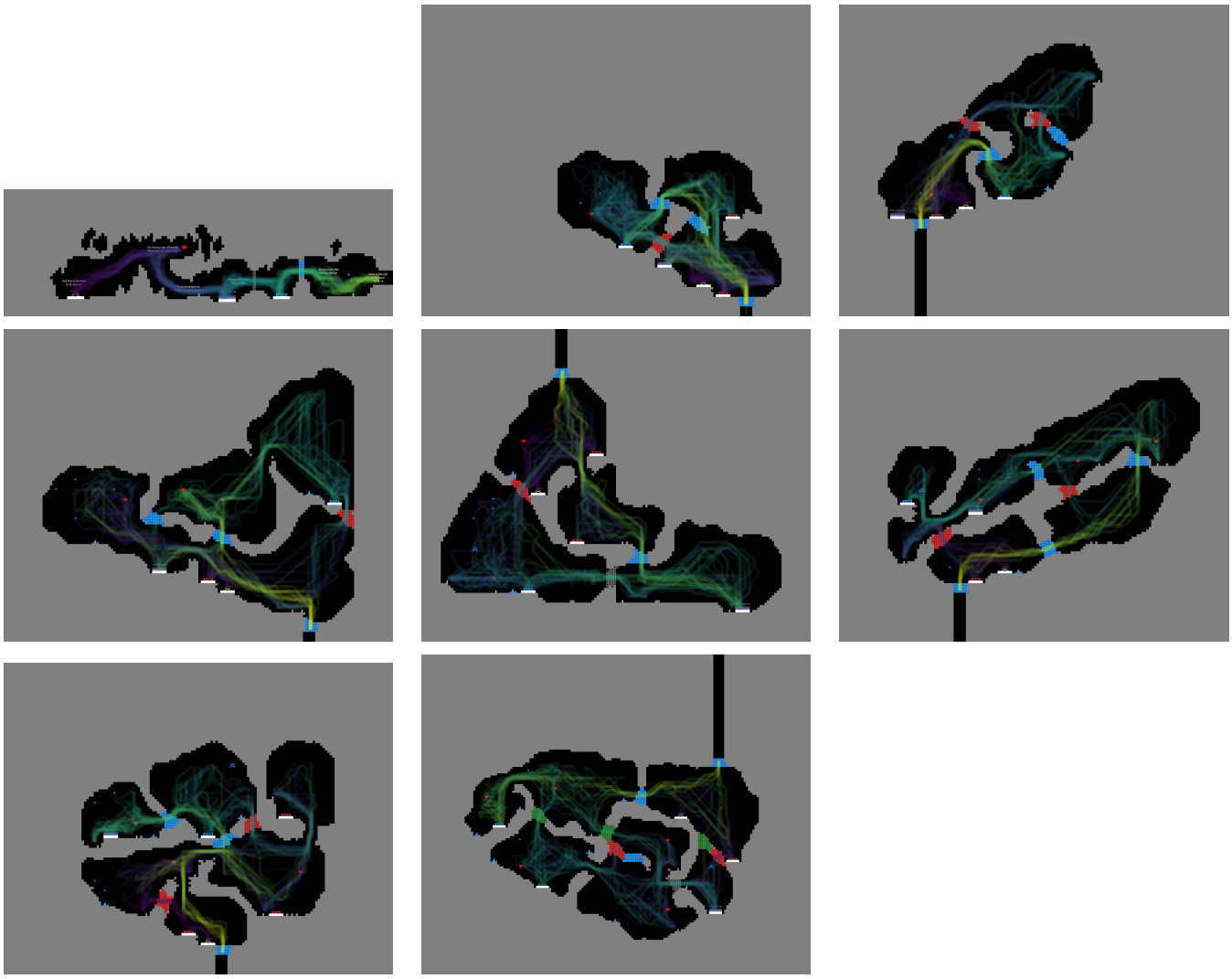

Table 5: All levels of the game Rescue Elite with an overlay of heatmaps showing the discovered solution paths. The heatmaps include only the solution paths taken by participants, starting at the beginning of the level (dark purple) and ending at the exit (yellow). The paths indicate that participants found multiple valid solutions.

A.3 Heatmap of Death Paths

Table 6 shows all the players' paths leading to the crash of the helicopter. The yellow circles indicate where a player took damage and the red crosses indicate where a player crashed or restarted the level. The death paths go from dark purple (start of the level) to yellow (point of crash).

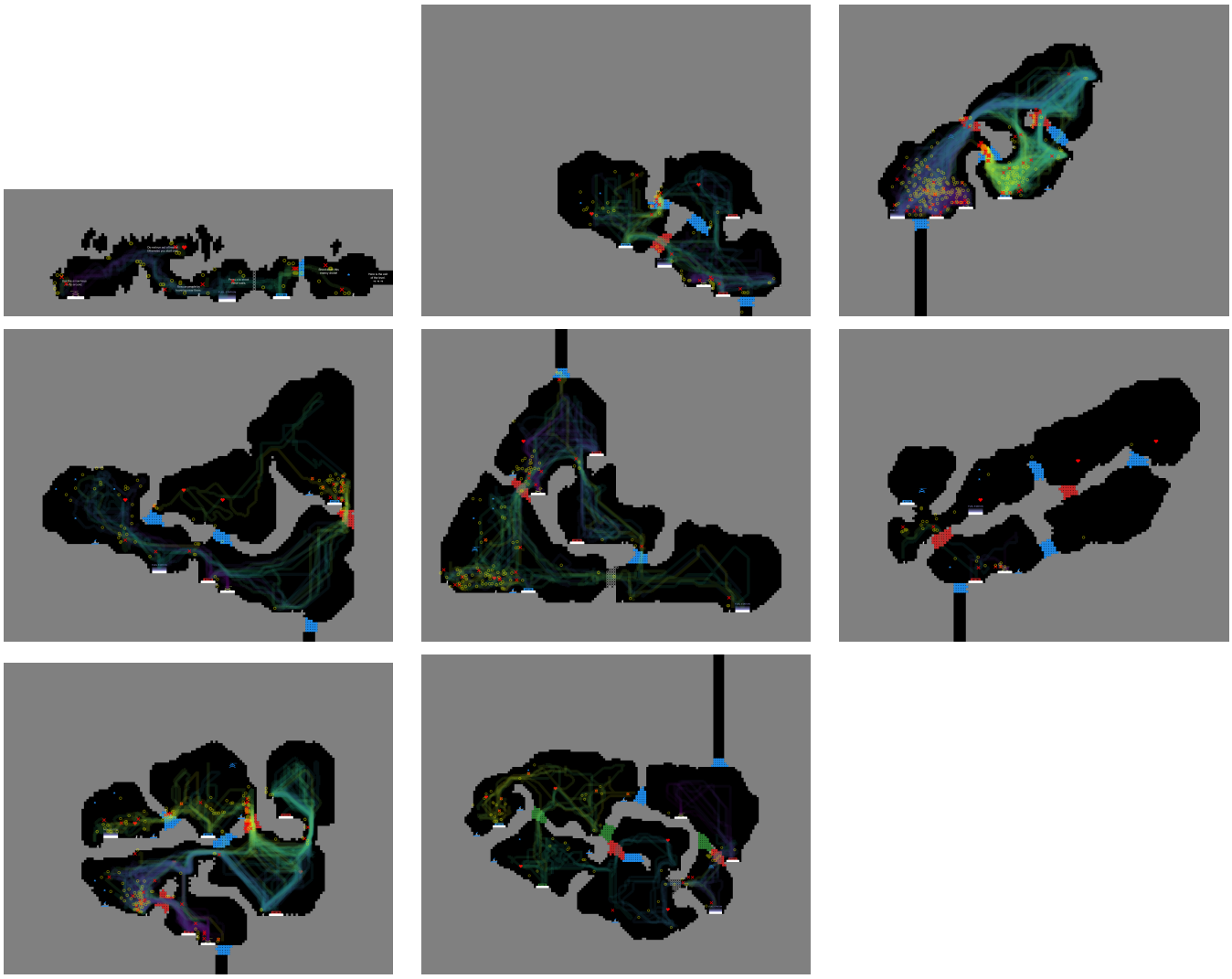

Table 6: All levels of the game Rescue Elite with a heatmap overlay showing paths taken until death. The paths start at the beginning of each level (dark purple) and end at the point of the crash (yellow). Yellow circles indicate where the helicopter took damage, and red crosses mark where the helicopter crashed.