A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels

ALEXANDER GELLEL, Australian National University, Australia
PENNY SWEETSER, Australian National University, Australia

Algorithmic generation of data, known as procedural content generation, is an attractive prospect within the game development industry as a means of creating infinitely fresh and varied content. In this paper, we present an approach to level generation for roguelike dungeon style levels, based on our examination of the suite of existing approaches used in formal research. Our generator aims to create simple dungeon style level layouts that are always playable. We utilise a hybrid technique combining context free grammars to generate a description of levels and a cellular automata inspired process to generate the physical space. The generator proves successful at consistently generating dungeon layouts that maintain completability at all times with sufficient variation, when accounting for the occasional need for corrective actions. We conclude that there is substantial value in hybrid approaches to automated level design and propose a new heuristic by which to assess dungeon style level content.

CCS Concepts: • **Applied computing** → **Computer games**; • **Theory of computation** → *Grammars and context-free languages*; • **Computing methodologies** → *Search methodologies.*

Additional Key Words and Phrases: procedural content generation, PCG, context free grammars, game design, dungeons, roguelike

ACM Reference Format:
Alexander Gellel and Penny Sweetser. 2020. A Hybrid Approach to Procedural Generation of Roguelike Video Game Levels. In *International Conference on the Foundations of Digital Games (FDG '20), September 15–18, 2020, Bugibba, Malta.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3402942.3402945

1 INTRODUCTION

Procedural content generation (PCG) is the algorithmic generation of data, as opposed to manual creation and input. PCG methods and tools that are easy to use, consistent, and reliable are a powerful way for game developers to create games that are replayable and enjoyable. PCG is not a new concept to the field of game design. The classic dungeon crawler, *Rogue* [39], is remembered fondly for its simple randomly generated maps and core gameplay loop that resets the player back to the start on the event of their failure. Rogue gave rise to the "roguelike" (sometimes "roguelite") genre of games, which is loosely defined by the use of randomised levels and permanent player death. The roguelike genre is the main application area for PCG in commercial games, due to the reliance of roguelikes on every play session being a fresh one. The majority of roguelike games have been created by smaller, independent developers and studios (e.g., *Binding of Isaac* [17], *Dungeons of Dredmor* [12]). The genre has also been explored as a training ground for AI [5], as it serves as an ever-changing environment.

There are many different methods and approaches that have been used for PCG in games, each with its own relative strengths and weaknesses. In this paper, we review and analyse the different approaches, techniques, and evaluation methods for PCG in games used in previous research. We identify that substantial opportunity lies in creating a hybrid

Manuscript submitted to ACM

approach, in which the combined techniques can minimise the weaknesses that exist for individual techniques in isolation. We present a prototype level generator for a non-specific roguelike dungeon crawler based on these findings, with a focus on ease of use and flexibility in its methods. Our mission generator combines a mission description generator based on context free grammars with a level space generator inspired by cellular automata that encodes varying non-deterministic rules. We also present and apply a new heuristic by which to assess dungeon-style video game levels.

This aim of the research presented in this paper is to create an effective and easy to use dungeon layout generator. The project focuses on levels for use in the context of a roguelike dungeon crawler game. While not designing for a specific game, many of the design traits for the dungeons (e.g., lock-key pairs, overall structure) are based on the style of classic adventure game *The Legend of Zelda* [21], as this best exemplifies the kind of structural rules that make dungeons an interesting content type to generate. We do not model more complex mechanics and gameplay systems, as these are game-specific and do not contribute to the task of generic level generation.

2 PCG IN GAMES

Games are often evaluated, at least partially, on their replayability, amount of content, and hours of play. However, quality game content usually needs to be manually constructed by designers and developers, which is both difficult and time consuming. PCG offers two primary gains to the field of game design. First, PCG offers the potential for nearly infinite content and variation for each play session. Recent high profile games, such as *Minecraft* [18] with its procedurally generated worlds and *Fortnite Battle Royale* [11] with its random distribution of weapons for players to pick up, have shown that randomness and frequent variation in game content are valuable traits for a game's longevity and success. Second, PCG offers the ability to create wholly unique experiences for a game. When more of the game is about interaction with game systems, rather than specific interactions created by the designer, players can create their own personal, or emergent, stories [32, 34]. This kind of systemic design also allows for the possibility of emergent gameplay, allowing for a broader range of actions, interactions, and strategies that might not have been planned or anticipated by the developer [32]. Game content with high degrees of randomness and variation often provides a strong basis for these unique experiences, as a playthrough can be defined by a lucky interaction between player and game.

PCG in games covers a wide range of problems and methods [36]. However, the scope of formal research into PCG methods is somewhat limited and game developers that utilise PCG often do not publish their methods or source code. Procedural generation is also not limited to playable content (e.g., maps, levels, rewards). It also has applications in the rendering of detail in natural looking environments. For example, SpeedTree [27] is a means of procedurally generating natural looking trees that is used in many existing games. Some of the techniques reviewed in this paper are not specifically used in level and map generation, but were examined for their general application and what value they can provide as a means of content generation. In this section, we review the approaches, techniques, and evaluation methods used for PCG in games.

2.1 PCG Approaches

Previous research has defined various distinctions between different PCG approaches. Togelius et al. [37] propose classifying PCG approaches according to (1) online versus offline, (2) constructive versus generate-and-test, (3) necessary versus optional, (4) random seeds versus parameter vectors, and (5) stochastic versus deterministic methods. They note that these distinctions are not binary, but rather should be considered as continuums.

- "Online" describes a generator capable of acting near immediately in real-time, capable of generating during play, making it important that the method is predictably fast and acts with sufficiently consistent quality. "Offline" generators are used between games and are more useful as an assistive tool for designers, with much less of a concern with predictability.
- A "constructive" approach is entirely dependant on its methods and rules to generate both without failure and to ensure that the resulting output will be of sufficient quality. Conversely, a "generate-and-test" approach is built in two parts, one method that generates content and another that evaluates it based on some relevant criteria. If sufficient criteria are not met, the content is disposed of and the generator is called once again until something more suitable is found. A "search-based" approach is also defined in Togelius et al. [37], which generates a large population of content and evaluates it by some heuristic to find a good or best desired instance.
- "Necessary" content must always work perfectly without fail, whereas "optional" content can be less strictly functional or optimal. The requirements for this will vary heavily with individual game design, but it is an important factor when considering how a level generator is to function.
- A "random seed" generator relies entirely on its generative rules to construct interesting variation, whereas more information offered via "parameter vectors" allows for higher degrees of configuration from players and designers and also opens up more possible analytical techniques such as utilising machine learning to find optimal balances of settings.
- The distinction between a "stochastic" and "deterministic" generation method is somewhat more nebulous, but asks whether or not giving identical starting data will lead to some given algorithm generating identical output results. Deterministic methods allow for a form of compression. For example, a level need only be saved as its seed if it can be guaranteed that a generation method can always unpack such a seed into the same resulting level.

Togelius et al. [37] also propose that hybridisation of approaches to suit specific problems could be quite productive and powerful. Dormans [7] defines two separate components of game level PCG. One is the physical "space", which represents the actual existing layout of a level, and the other is the "mission", which describes the sequence of tasks a player must complete in order to successfully complete the level. These two pieces of a level are mapped to one another but otherwise exist wholly distinct.

2.2 PCG Techniques

Numerous techniques have been utilised for PCG in both research and commercial games. Table 1 presents a side-by-side comparison of the strengths and weaknesses associated with the different techniques. Many approaches have strengths that oppose the weaknesses of others, which would suggest that a hybrid approach could effectively combine complementary techniques to minimise weaknesses.

2.2.1 Markov Chains. Markov Chains and Random Markov Fields have been used to generate levels in classic 2D side scrollers [26], including *Kid Icarus* [20], *Lode Runner* [25], and *Super Mario Bros.* [19]. In Snodgrass and Ontañón [26], multi-dimensional Markov Chains were observed to create aesthetically pleasing levels, sampling in only a fraction of a second, making them suitabile for use as an online form of generation. However, low success rates for *Loderunner* and *Kid Icarus* and less than total playability for *Super Mario Bros.* suggests that such models are not sufficient as a means of generation for platformer type levels on their own, due to the model not being able to encode specific knowledge

Table 1. Comparison of techniques for PCG in games

Technique	PCG Strengths	PCG Weaknesses
Markov Chains and Random Markov fields [26, 31]	Fast, aesthetically varied, outputs resemble statistically similar distributions of components.	Reliant on statistical properties on a local scale, fails to see long term dependency. Needs active repair utilising domain knowledge.
Cellular Automata [15]	Fast, simple, low cost. Convincingly natural environments.	Unstructured, chaotic. Lack overarching control or long term understanding.
Grammars [6–9]	Structural rules allow for lots of control. Definable terminal elements allow for easy substitution and enforcing of important features.	Rules are not versatile without manual changes.
Machine Learning [16, 23, 29, 30, 38]	High variety, capable of capturing statistical patterns of human-authored levels. Bayes networks can recognise higher level features.	Often needs repair of unplayable sections. Requires extensive dataset, suitable for existing games not new ones. Game content has underlying rules that statistical analysis tends not to capture.
Evolutionary Algorithms [1, 3, 10, 13, 35, 41]	Naturally finds optimal solutions, high variation as a product of recombination.	Highly dependant on meaningful fitness functions to separate good from bad. Can get stuck in local optima.
Player Experience Models [9, 23, 43]	Configurable to player interests or in response to player evaluations.	Dependant on existing configurable generation methods.

about the rules of the game that allow it to be playable. Summerville et al. [31] used Monte Carlo Tree Search to guide Markov Chain generation.

2.2.2 Cellular Automata. Cellular automata (CA) are spatial, discrete time models represented on a uniform grid, which can be used to model different aspects of game environments [33, 34]. CA have been used to generate infinite and natural looking game cave spaces as compared to a purely random method [15]. In Johnson et al. [15], the generation times were very fast, with both random and CA models generating formations in less than a millisecond each, making them suitable for online types of generation. However, the spaces were carved out without structure or purpose and would require additional methods to define meaningful content.

2.2.3 Grammars. Generative grammars have been explored in the PCG space for a few uses. Grammars are linguistic based models consisting of a set of terminal and non-terminal symbols, which are traditionally used in code parsers to validate syntactical correctness. For PCG, grammars offer a means of structured recursive generation with strict rules to define what is and is not possible. Grammars are highly suitable for generating an intended sequence of *necessary* actions for a player to take, such as the game missions. Generative grammar models have been used to generate quests in the style of existing MMOs [6]. In Doran and Parberry [6], the substitution nature of grammatical structures were

observed to allow for easy pseudo-random selection of components (e.g., characters to talk to, locations to visit, items to find).

Models of generative grammars have also been proposed as an approach to missions and spaces [7, 8] and mapped to player experience models [9]. In these papers, the authors used graph grammars to generate branching substructures of optional content, linear lock-key pairs, and scattered multi-piece keys required to bypass single locks. The content was then mapped to a space using space grammars, preserving the structure of the original graph as a physical level. An example of a simple grammar that defines the structure of a basic *Legend of Zelda* [21] style dungeon is defined in Dormans [7]:

```
1. Dungeon -> Obstacle + treasure
2. Obstacle -> key + Obstacle + lock + Obstacle
3. Obstacle -> monster + Obstacle
4. Obstacle -> room
```

The primary strength of grammars as a generative method is that they offer rigid structure with a natural hierarchy that is easily enforced, while terminal components can easily be substituted for the important features of a target game. This is important for level generation that requires any kind of intended sequence or has properties that are not easily represented by a simple statistical pattern. Dormans' model of Mission and Space [7, 8] is particularly effective, as it offers a means of structuring content in a way that other methods do not.

2.2.4 Machine Learning. Machine learning (ML) techniques are similar to Markov Chains, in that ML is heavily reliant on statistical properties found within datasets of existing manually designed game levels. In Summerville and Mateas [30], ML was used to generate levels for *The Legend of Zelda: A Link to the Past* [22]. A number of fixed rooms were first placed based on the learning of a Bayesian network capturing various high level features. The rooms were then populated by taking two samples from a dataset and interpolating them. It was found that this process did not guarantee the generation of completely playable rooms. The ML family of approaches are strong in their performance, but are dependent on access to strong datasets and meaningful evaluation functions that may not always be available, making them unsuitable for PCG systems being built from the ground up.

In Torrado et al. [38], conditional Generative Adversarial Networks (GANs) were used to create simple single-room content for *The Legend of Zelda*, targeting a single lock-key pair and basic enemy distribution. It was observed that while GANs are successful in generating simple content (e.g., images), they tend to fail to capture the structural rules that underlie a well designed game level. The playability rate was at most 58% with a training set of 45 levels, which is unsuitable for a generative method and even somewhat unreliable for a search-based method without a much larger dataset.

The biggest weakness of ML for PCG is the need for large datasets. If a game designer wishes to use ML in their own game, they would first need to create a large number of valid levels. Torrado et al. [38] developed a bootstrapping mechanism to address this problem, while Khalifa et al. [16] trialled Reinforcement Learning as an alternate approach. ML does have other uses in games beyond level generation. If an existing content generator is available, ML can be used to configure the settings, such as size and difficulty. In Roberts and Chen [23], a proof of concept level generator was constructed for *Quake* [14] that could adjust its parameters based on a set of 122 features provided by output logs. This was evaluated against existing models of player profiles, allowing the game to adjust its difficulty and content types to suit the player. Their preliminary results showed promise.

2.2.5 Evolutionary Algorithms. Evolutionary algorithms, which are a class of optimisation algorithm, are a growing field within the study of PCG. Evolutionary methods have been used to model 3D terrains [10], optimise strategy game maps [35], select appropriate levels [3, 41], and co-create dungeons [1]. Frade et al. [10] were successful in generating varied terrains suitable for both player traversal and feature placement. However, some of the generated terrains appeared chaotic and unnatural, as realism was not a focus. Togelius et al. [35] optimised strategy game maps by use of multi-objective evolutionary algorithms that selected and recombined maps from a random population. Their fitness functions looked for density of resources, distances between player bases, and factors such as chokepoints. They found that some elements of good content naturally conflict with each other, such that evaluation functions need to be meticulously crafted to maximise these features. In Ashlock [3], Tournament Selection was used to identify the best maze-like levels generated based on a set of four fitness functions. The performance of the population was found to increase with the generations and the mazes evaluated on the different fitness functions generated in varied ways. In Valtchanov and Brown [41], similar results were observed with their evolution of dungeon levels. With the appropriate domain-specific fitness functions, evolutionary algorithms provide a powerful means of selecting good levels from a search-based population. Evolutionary algorithms offer a lot of promise in the field of PCG, as they handle broad random populations and naturally reach solutions deemed optimal by their fitness functions. However, this is also their biggest drawback, as they need a clearly defined fitness function (or set of functions) against which content can be evaluated. This requires quantifying good game design into a set of objective rules, which not only varies by game but can also struggle to capture the essence of an intended experience when abstracted to such a degree.

2.2.6 Player Experience Models. Once existing generative methods are available, they can be configured against models of player preferences. Yannakakis and Togelius [43] have investigated the use of player profiling to model preferential experiences and configure level parameters to those models. In their work, a player's experience can be modelled both directly and theoretically in a variety of ways, proposing that such a system could be utilised for real-time adjustments to the quality of game content to personalise a game experience. These methods are promising but require existing generation systems to handle the core process, which can then be configured by such profiles.

2.3 PCG Evaluation

All games have their own rules and restrictions, as well as many subjective quantifiers of quality for generated content. As such, we propose that the only universal evaluation metric for generated levels is whether or not they are physically completable or playable. [26] achieved as high as 90.4% completability with their Markov Chain models for a set of 1000 levels generated within *Super Mario Bros.* However, for any value less than 100%, a level generator must always be prepared to discard or actively repair the generated content.

One solution is to have human players playtest generated levels. However, this would not help a generator assess its own content actively, as would be needed in any search-based approach. [30] used a Bayesian Information Criterion to compare levels to a set of human-authored levels, but this requires an existing dataset for comparison. [3] devised unique fitness functions to measure the "interestingness" of generated content based on various factors. However, such functions are usually bespoke to the specific content produced and are not universally applicable to content generators.

Researchers have lamented the lack of universal measures by which to compare the quality of levels generated by PCG. Modelling the processes of content generation more formally within a framework as in Amato and Moscato [2] could also help lead to a more universal system of designing content generation. However, most game creators will likely adhere to their own unique designs dependent on a specific game's needs.

2.4 Summary

In this section, we have examined various approaches to PCG in game design research and how these approaches can be evaluated, defined, and analysed. Each of these approaches has limitations on the type or quality of level content they can produce and/or the requirements for producing the content. In this paper, we present a prototype simple level generator that combines the structural rules and design potential provided by generative grammars with the simplicity and speed of cellular automata-style behaviour in order to map it to a physical space. We also define a unique evaluation metric suited to the types of levels we generate.

3 DESIGN

In this project, we aim to hybridise two of the approaches to PCG discussed in the literature review to produce a simple and easily controllable dungeon-style level generator. The focus of the generator is on the distribution of *necessary* content and consistency in performance. Our chosen level style for this generator is a classic 2D top-down dungeon, inspired by the roguelike genre and functionally similar to *The Legend of Zelda*.

Dungeons make for interesting test bases for level generation due to their structure and data representation. In terms of structure, dungeons are neither linear (i.e., a path of *necessary* objects) nor open world (i.e., a single open space to freely wander about). Typical dungeons offer a definite path to completion, but finding that path often requires exploration. Dungeon designers must impose structural rules to define where a player can explore at a given time, while ensuring that the spaces are sufficiently complex to test a player's spatial awareness and navigational memory. This tends to be achieved by lock-key pairs or an equivalent prerequisite-obstacle system disguised by narrative or mechanics (e.g., weapon-material, level-barrier, equipment-terrain). These conditionally passable obstacles break the level into subsections, such that the player must explore the level and solve puzzles and/or fight enemies until they find a key that grants access to another section. Consequently, for a level to be playable, all keys must be reachable before their intended locks and their generation as pairs is *necessary* content. Our generator focuses on producing levels to ensure that subsections are controlled and completable.

In terms of data, dungeons are easy to represent. Dungeons in games like *The Legend of Zelda* or the roguelike *The Binding of Isaac* [17] consist of a grid of equally sized, distinct, adjacent rooms. Grids are typically an ideal representation of data because they are easy to define. Each room has its own content and a set of open or closed doors that allow movement to connected rooms. Some doors will be locked and require a key to progress. Rooms can also be prefabricated as a component, allowing them to be conjoined and arranged independently of their content and streamlining the process of level construction.

3.1 Context Free Grammars

We selected generative grammars to define missions, similar to those described by Dormans [7, 8]. We use the grammar rules to recursively generate a single string, which is used to describe a flow and intended order of room types that a player will visit. The grammar rules enforce the generation rules, so that only valid missions are generated. This process is also very fast and can be used rapidly to treat this as a search-based problem. In the context of dungeon style levels, these top level rules enforce any *necessary* structure that may be otherwise overlooked by the space generation methods. For instance, we start each generation with a rule defining a start and end point and recursively filling in the content:

```
Dungeon -> start + Content + end
```

To enforce higher level rules, such as a minimum number of rooms, or even to ensure that a set of important rooms occurs in sequence, we can define:

```
Dungeon -> start + room + room + importantroom1 +
  Content + importantroom2 + end
```

This approach can also force all keys to generate in appropriate pairs with a lock, such that no lock exists without a corresponding key, whilst also ensuring that every key appears before a lock in all circumstances. This is achieved by only defining rules that can generate keys in such a way that this order is maintained. In this context, keys are universal to any lock rather than bound to a specific lock, so by recursively generating a new lock-key pair within another one, the new lock does not directly impede the progress on reaching the original one that generated first. As such, there is always going to be one or more keys before any lock with a simple definition:

```
Content -> Content + key + Content + lock +
  Content Content -> room
```

The terminal values can be easily substituted. For instance, a lock-key pair could be replaced with a game specific item, such as a jetpack and a bottomless pit to cross. All that matters here is that the lock component is considered to be impassable without the appropriate key. No particular generative rule set is emphasised in this paper, as the intent is for this generator to be configurable to the needs of a generic game. A single rule set will be used in most situations for consistency. Here is a simple recursive grammar designed to output similar but varied descriptions for the space generator:

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

3.2 Cellular Automata-Inspired Behaviour

We selected a cellular automata (CA) inspired approach to generate the physical spaces that complement the mission descriptions. Behaviour based on CA is ideal, as it affords both speed and a natural-feeling distribution of constituent components. CA can produce natural spaces with an element of chaos but without structure and order, which the strict rules of the grammar system can provide. Additionally, as our dungeons are represented as a grid, they can be modelled as cells without any additional alteration. Our approach is CA-inspired, but is not a traditional CA due to the use of non-determinism and information being provided beyond the scope of a cell's immediate neighbourhood.

The generator is given the mission description in the form of a single string, as generated by our grammar described in the previous section. This approach also allows a designer or player to submit a custom string to the generator if they wish to create a specific scenario. In order to keep the space generation generic, we aim to encode as little higher level understanding of the structure of the actual space as possible. This string controls the size and the overall distribution of content within the level, independent of the space generator. The lock is the defining feature separating subsections of the level from each other. The generator places each room one at a time and in the order described by the initial string, following simple but pseudo-randomised rules. At any given time, a single room is considered the current active cell and the generator will always attempt to move to an empty cell to place the next room.

3.2.1 CA Rules. There are two main variants of the rules, described as follows. In the first rule (see Figure 1), the generator is given the location of its start room; it then selects a pseudo-random cell from its immediate neighbourhood that is a valid move (i.e., not a room); it then moves to it to place the next room, which is then marked as the current type in the mission string. The generator maintains no further knowledge of the entire grid in this process. This process is repeated for each symbol in the mission description until it reaches the end room or until the program is incapable of making a move. This simple method serves as a baseline of completability to compare to others.

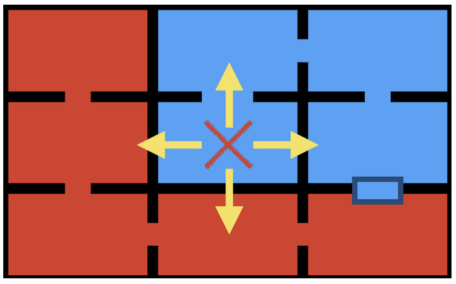

Fig. 1. The first CA rule. Subsections shown in red/blue, X denotes active room, and arrow search directions. This scenario shows no valid options for room placement.

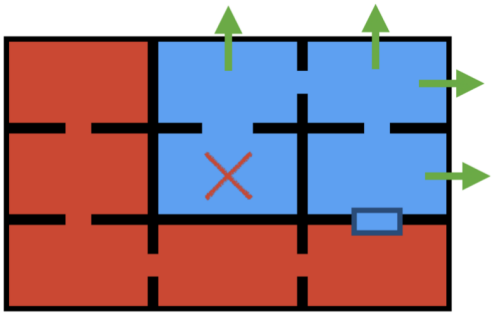

Fig. 2. The second CA rule. Subsections shown in red/blue, X denotes active room, and arrow search directions.

In the second rule (see Figure 2), the generator attains knowledge beyond the single active cell. For any given subsection of the map, the generator evaluates its surroundings and picks the first random valid direction branching off from the room that is closest to that section's average centre point as it can reach, based on Manhattan distance. We term this a Subsection Search and it allows the space to appear as though it is growing around a natural cluster, rather than tracing a line while leaving a trail.

3.2.2 Search Methods. We propose two approaches to using these rules. In one approach, which we term a Persistent Subsection Search, the generator will use the second rule at all times. In the other approach, which we term a Subsection Search on Halt, the first rule will be used until a dead end is encountered, in which case the second rule will be employed. One final rule is added to allow the generator to overwrite its current location should no possible moves exist, even after a subsection search. In this third rule, placement of an end is forced, replacing an unimportant room. This rule is

equivalent to repairing and acts as a failsafe to ensure completability above all else, as the central attribute of successful level generation. A flag is marked during generation to indicate if this has occurred.

After the topology is generated, the generator will go back and iterate over all the rooms and open pseudo-randomly selected doors between rooms of the same subsection based on an arbitrary probability. A common approach to dungeon level generation is to place a few separate rooms and "drill" tunnels between them to create a more cohesive cluster of spaces [24]. This approach allows the subsections to open up for exploration without having to encode in higher level patterns, as well as to provide a means to force totally open or totally constrained subsections at the extremes.

3.3 Path Difference Heuristic

As no universal measures for quality of video game levels exists, we devised a bespoke evaluation heuristic for comparing the approximate "interestingness" of the generated content, for both differentiating the two CA rules, but also for potential use in a search-based design. In the case of a dungeon style level, we want a way to measure the semi-linearity property as discussed earlier, to evaluate levels for being both structured and encouraging exploration at the same time (see Figure 3).

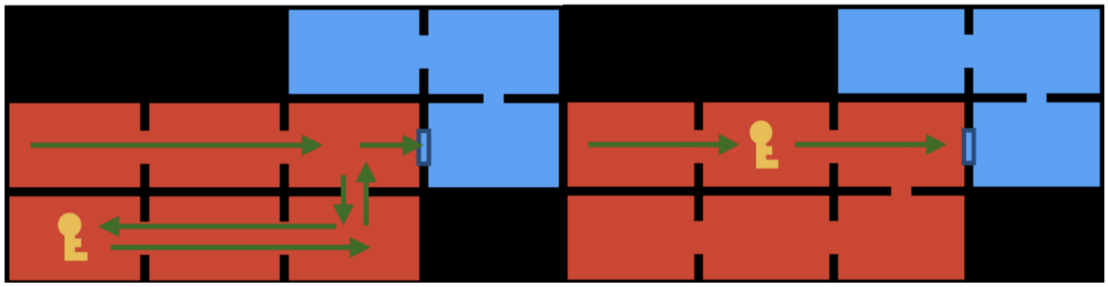

Fig. 3. The interestingness of a generated dungeon. Left section shows a more interesting path, whereas the right section shows a more direct path.

Our heuristic considers two possible paths through a dungeon. The first is the Critical Path, which is the shortest path required to complete the level by attaining all keys and taking them to all locks to reach the end. The second path, which we call the Spine, details the same path but does not factor in the journey required to find any of the keys. The spine represents the shortest possible path that could occur between the start and end, if all keys and locks were to fall along the way in the right order. The heuristic used by the generator assesses levels based on the total difference in length between these two defined paths. The Path Difference heuristic is defined as:

$$Length_{Critical} - Length_{Spine}$$

The larger the Path Difference, the more a player needs to explore the map to find the *necessary* keys to traverse it to completion. Assessing the Critical Path length alone would encourage selection of a single long winding path, which would make for a more linear level. By contrast, the difference between the two proposed paths rewards short Spines with spaces branching off them. Furthermore, this subtractive value helps to normalise the resulting value across differently sized dungeons. Figure 4 visualises the difference in the two paths. As all the rooms will have an index

in order of their generation, this allows the keys and locks to be assessed in their intended order with point to point pathfinding over the 2D space.

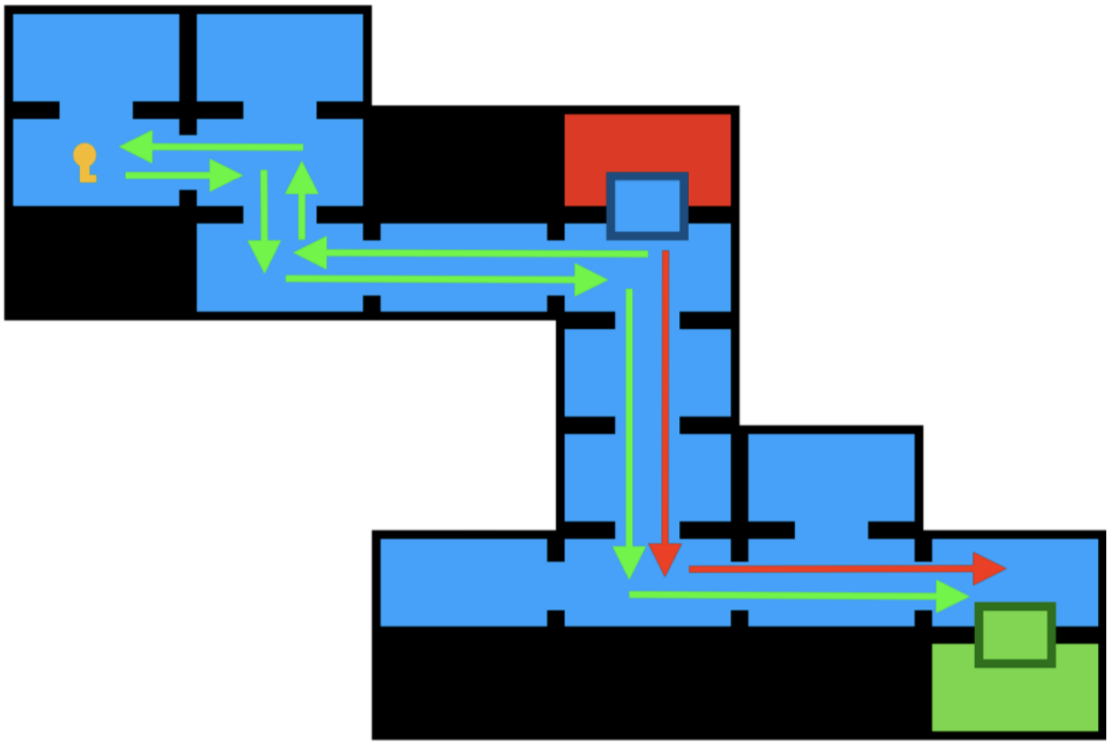

Fig. 4. The Path Difference for a subsection. Critical Path in green arrows (length = 14), Spine in red (length = 6). Subsections shown in red/blue/green.

3.4 Implementation

The generator is assembled separately in its two parts. The mission generator is built on a simple technique and produces its output as strings. It is implemented as a Python script sentence generator based on [4], using a convergence rule to help limit the recursive pattern from generating forever and then modified with the dungeon generation rules presented earlier. It is fast and can be easily configured to produce a large population of missions in a search-based manner.

The space generator is constructed in the Unity Game Engine [40], a common development tool for games across many platforms, using C# as its primary scripting language. The dungeon generator program contains a core class that handles the generation methods and produces a large, finite, 2D grid of rooms. Each room is defined as a class that has properties detailing its type (inherited from the mission description), the status of its doors, and the colour of its containing subsection.

Table 2. Average mission generator runtimes (seconds).

100 Missions	1000 Missions	10000 Missions
1.09×10^{-2}	1.05×10^{-1}	1.07

4 EVALUATION

Any constructive approach to level design is insufficient if it fails to achieve 100% playability of its levels, where search-based designs are allowed some leniency. As no standardised benchmarks exist for PCG at present, the generator and its variations will be analysed in terms of its:

- Ability to produce levels that are possible to play to completion; and
- Comparative performance of the two more refined search methods with respect to the defined heuristic.

4.1 Grammar Generator Performance

Although the quality of the generated missions is difficult to assess, the performance of the generative method and basic selection of good candidates is sufficiently fast for large quantities of produced missions to be suitable for any usage. A timestamp was taken at the beginning and end of runtime, across 10 samples for 100, 1000, and 10000 productions (see Table 2). Time records were taken using Python's internal timestamps, printed as an output, and evaluated externally. As for the properties of the generated mission descriptions, due to the structure of grammatical rules, 100% of lock-key pairs were generated appropriately and the start and end always fell at the first and last symbols, respectively. These were evaluated automatically across numerous sample sets without fail.

4.2 Space Generator Performance

Each space generation rule was tested for 500 generated levels and automatically evaluated both for completability and against the Path Difference heuristic using pathfinding between the critical points. The Random Neighbour Search variant of the program simply reported back a False value whenever it hit a point where it could no longer progress and the Subsection Search variations reported whether they had to force the generation to come to its end.

All tests were conducted on a single manually selected mission description, consisting of 58 rooms and 8 lock-key pairs. This was created specifically to be somewhat larger than those typically selected by the grammar generator, in order to increase the likelihood of collisions in the space generation behaviour. The chance for a door between adjacent rooms of the same subsection to be opened was set to 25%[1]. Additionally, the tests were only carried out using four colours to distinguish subsections, instead of a possible eight. This allowed for a potential fusion of subsections due to the door opening to observe if that impacts the physical playability of the space. Results of the Path Difference Heuristic (see Figure 5) were collected alongside completability information for the two variants utilising subsection searches (see Table 3).

4.3 Space Generator Observations

The Random Neighbour Search did not have sufficient complexity to generate complete levels consistently and it halted generation on a number of tests. A halt occurred when all four neighbours of a room had previously been set, which occurred frequently, particularly on generations with a large number of rooms. The Persistent Subsection Search (see

[1]As this is applied on both sides of a door, in practice the chance is 7/16 or about 44%.

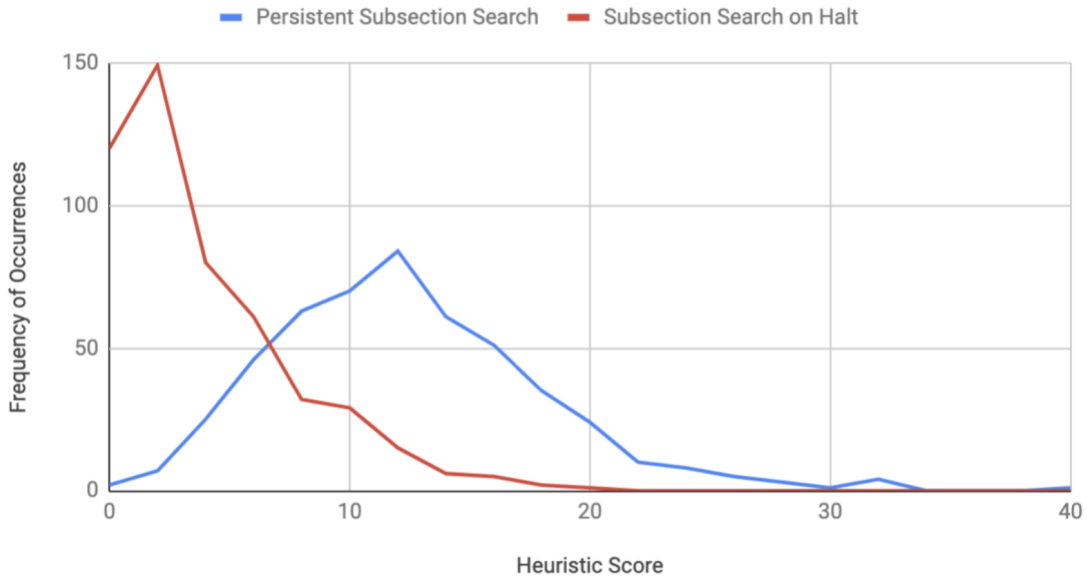

Fig. 5. Path Difference heuristic performance between the two methods (higher score means more exploration).

Table 3. Space generator completability results.

Random Neighbour Search	Subsection Search on Halt	Persistent Subsection Search
53.2%	100% (20% forced end)	100% (7% forced end)

Figure 6) appeared to generate a more varied distribution of rooms than the Subsection Search on Halt (see Figure 7), with fewer long uninteresting corridors that lacked freedom to explore. Generated levels appeared to be completable at all times, based on lock-key distribution. Only occasional generative defects occurred that required the generator to force an end to ensure completability or when a subsection of the same colour fused with another to create a central branching area.

5 DISCUSSION

In this paper, we reviewed the approaches, techniques, and evaluation methods for PCG in games. Based on our review, we proposed a hybridised approach to PCG of game levels that we tested in a roguelike dungeon style level generator. Our aim was to create a simple and effective dungeon generator, based on generative grammars and cellular automata inspired behaviour. We also proposed a new heuristic based evaluation method that we used to evaluate our generator, in addition to checks for completability and performance. For our generator, we proposed, tested, and compared different variations of our basic generation rules.

The baseline approach (Random Neighbour Search) was only able to generate completable levels about half of the time (53.2%). Without additional high-level knowledge or more complex rules to safeguard it, the generator was often unable to place an end room and create a playable level. The other two methods, with their wider pool of available actions, did not encounter generative failures and were always able to generate a playable level. In all generations, all locks and keys existed in accessible pairs on the map, even despite some generative defects. This result included the reparative forced end rule, meaning that some rooms in the mission description were deleted, ensuring that the level remained playable.

Between our two approaches, the Persistent Subsection Search achieved a higher score on the Path Difference heuristic. This is to be expected given the constant freedom of growth it is provided compared to the more linear trail left by its simpler counterpart. This is best emphasised by 24% of the simpler levels having a Path Difference score of 0, meaning that the player would not need to venture off the Spine path in their journey to the end. Of course, whilst a player is not always guaranteed to take the right path first try, this suggests the formation of long corridors with little variation or option to digress. Conversely, only 0.4% of the Persistent Subsection Search levels had no difference at all between paths.

We can conclude that the Path Difference heuristic functioned as intended, scoring higher on levels that appeared less linear in their overall layout. Ashlock et al. [3] observed when assessing their evolution of maze-like maps that a fitness function based solely on maximising the distance between the start and the end points encouraged selection of long and uninteresting winding paths. Our heuristic achieved the opposite, scoring highest on levels with short absolute distances from the start to end, but requiring more exploration around the space to actually reach it. Our Persistent Subsection Search approach could be further enhanced in use as a search-based method to remove levels below a certain threshold or the small number of levels that required some generative protection to ensure completability.

The aesthetic results of our generator are inline with similar experiments in the field, producing varied dungeon layouts that feel both familiarly structured yet varied. The performance of the generator makes it sufficient to function as a constructive PCG method, when factoring in the ability to prevent levels from being unplayable during generation. However, our heuristic also affords the option of being used as a search-based approach. In theory, search-based methods will always produce more consistent results, as they can select better content built by the rules and account for any generative defects. However, search-based approaches are slower and may not be suitable for use as a level generator online during a game.

Our results suggest that hybridised generative techniques are an effective means of producing interesting procedurally generated content that are worthy of further investigation and refinement. Our hybrid method produced usable results with ease, while making use of simple rules and only requiring some repairs during generation. Our two selected techniques were able to effectively complement each other's generative weaknesses, with the grammar rules providing a structured order that was not too rigid and the non-deterministic automata providing varied and sufficiently random spaces.

5.1 Limitations and Future Work

The dungeons generated in this paper are designed as top-down 2D levels, but they have been configured as a 3D model within the Unity engine. 3D dungeon levels are common in RPGs such as *The Elder Scrolls V: Skyrim* [28] and are far more complicated to model than the kind addressed in this work. A natural next step would be to expand our generator to change the elevation of rooms to create an explorable 3D space. However, current attempts to generate dungeons in 3D space [42] tend to feel much more unnatural than their 2D counterparts, especially with regards to connecting

rooms at different elevations. Additionally, our generator produces its subsections in a mostly linear manner, assuming that one locked off section naturally follows another. Another evolution to the design would be to create branching tree-like subsections that can be completed in any order. However, this would require additional rules to ensure the dungeon is always completable under all possible orderings and that generated content remains balanced.

6 CONCLUSION

In this paper, we analysed existing approaches and techniques used for PCG in games, emphasising their relative strengths and weaknesses and their suitability for use as a basis for PCG in new games. Based on this analysis, we devised a prototype dungeon style level generator that used a hybrid approach of generative context free grammars and behaviour based on cellular automata. Our generator achieved a high completability rate for generated levels and total completability with the addition of failsafe rules, along with aesthetically varied results and patterns. We also presented a new heuristic for evaluating 2D dungeon levels that can be used for search-based methods and dungeon style content in general. Our heuristic attempts to capture the semi-linear qualities of dungeon style levels and proved effective at measuring this concept in our tests. Our hybrid approach to dungeon generation provides both user control and sufficient randomness for generated levels to feel different on successive generations. The ease at which the hybrid approach produced effective results is a promising direction for future approaches to PCG in games.

REFERENCES

[1] A. Alvarez, S. Dahlskog, J. Font, and J. Togelius. 2019. Empowering Quality Diversity in Dungeon Design with Interactive Constrained MAP-Elites. In *2019 IEEE Conference on Games (CoG)*. 1–8.

[2] Flora Amato and Francesco Moscato. 2017. Formal Procedural Content Generation in Games Driven by Social Analyses. In *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 674–679. https://doi.org/10.1109/WAINA.2017.3

[3] Daniel Ashlock, Colin Lee, and Cameron McGuinness. 2011. Search-Based Procedural Generation of Maze-Like Levels. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (Sep. 2011), 260–273. https://doi.org/10.1109/TCIAIG.2011.2138707

[4] Eli Bendersky. 2010. *Generating random sentences from a context free grammar.* Retrieved January 21, 2020 from https://eli.thegreenplace.net/2010/01/28/generating-random-sentences-from-a-context-free-grammar/

[5] Vojtech Cerny and Filip Dechterenko. 2015. Rogue-Like Games as a Playground for Artificial Intelligence – Evolutionary Approach. In *Entertainment Computing - ICEC 2015*, Konstantinos Chorianopoulos, Monica Divitini, Jannicke Baalsrud Hauge, Letizia Jaccheri, and Rainer Malaka (Eds.). Springer International Publishing, Cham, 261–271.

[6] Jonathon Doran and Ian Parberry. 2011. A Prototype Quest Generator Based on a Structural Analysis of Quests from Four MMORPGs. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (Bordeaux, France) *(PCGames '11)*. Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages. https://doi.org/10.1145/2000919.2000920

[7] Joris Dormans. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (Monterey, California) *(PCGames '10)*. Association for Computing Machinery, New York, NY, USA, Article 1, 8 pages. https://doi.org/10.1145/1814256.1814257

[8] Joris Dormans. 2011. Level Design as Model Transformation: A Strategy for Automated Content Generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games* (Bordeaux, France) *(PCGames '11)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2000919.2000921

[9] Joris Dormans and Sander Bakkes. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (Sep. 2011), 216–228. https://doi.org/10.1109/TCIAIG.2011.2149523

[10] Miguel Frade, F Fernandez de Vega, and Carlos Cotta. 2010. Evolution of artificial terrains for video games based on obstacles edge length. In *IEEE Congress on Evolutionary Computation*. 1–8. https://doi.org/10.1109/CEC.2010.5586032

[11] Epic Games. 2017. *Fortnite Battle Royale.* Game [Windows]. Epic Games.

[12] Gaslamp Games. 1991. *Dungeons of Dredmor.* Windows, Mac, Linux. Gaslamp Games.

[13] D. Gravina, A. Khalifa, A. Liapis, J. Togelius, and G. N. Yannakakis. 2019. Procedural Content Generation through Quality Diversity. In *2019 IEEE Conference on Games (CoG)*. 1–8.

[14] id Software. 1996. *Quake.* Game [MS-DOS]. GT Interactive.

[15] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. 2010. Cellular Automata for Real-Time Generation of Infinite Cave Levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (Monterey, California) *(PCGames '10)*. Association for Computing

Machinery, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/1814256.1814266

[16] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. 2020. PCGRL: Procedural Content Generation via Reinforcement Learning. arXiv:cs.LG/2001.09212

[17] Edmund McMillen. 2011. *The Binding of Isaac.* Game [Windows].

[18] Mojang. 2011. *Minecraft.* Game [Windows]. Mojang.

[19] Nintendo. 1985. *Super Mario Bros.* Game [NES]. Nintendo.

[20] Nintendo. 1986. *Kid Icarus.* Game [Famicon]. Nintendo.

[21] Nintendo. 1986. *The Legend of Zelda.* Game [Famicom]. Nintendo.

[22] Nintendo. 1991. *The Legend of Zelda: A Link to the Past.* Game [SNES]. Nintendo.

[23] Jonathan Roberts and Ke Chen. 2015. Learning-Based Procedural Content Generation. *IEEE Transactions on Computational Intelligence and AI in Games* 7, 1 (March 2015), 88–101. https://doi.org/10.1109/TCIAIG.2014.2335273

[24] RogueBasin. 2009. *Grid based Dungeon Generator.* Retrieved January 21, 2020 from http://roguebasin.roguelikedevelopment.org/index.php?title=Grid_Based_Dungeon_Generator

[25] Doug Smith. 1983. *Lode Runner.* Game [Famicom]. Broderbund.

[26] Sam Snodgrass and Santiago Ontañón. 2017. Learning to Generate Video Game Maps Using Markov Models. *IEEE Transactions on Computational Intelligence and AI in Games* 9, 4 (Dec 2017), 410–422. https://doi.org/10.1109/TCIAIG.2016.2623560

[27] SpeedTree. 2020. *SpeedTree.* Retrieved January 21, 2020 from https://store.speedtree.com/

[28] Bethesda Game Studios. 2011. *The Elder Scrolls V: Skyrim.* Game [Windows]. Bethesda Softworks.

[29] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural Content Generation via Machine Learning (PCGML). *IEEE Transactions on Games* 10, 3 (Sep. 2018), 257–270. https://doi.org/10.1109/TG.2018.2846639

[30] Adam James Summerville and Michael Mateas. 2015. Sampling hyrule: Multi-technique probabilistic level generation for action role playing games. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.*

[31] Adam James Summerville, Shweta Philip, and Michael Mateas. 2015. MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.*

[32] Penny Sweetser. 2008. *Emergence in Games.* Nelson Education.

[33] Penelope Sweetser and Janet Wiles. 2005. Combining Influence Maps and Cellular Automata for Reactive Game Agents. In *Intelligent Data Engineering and Automated Learning - IDEAL 2005*, Marcus Gallagher, James P. Hogan, and Frederic Maire (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 524–531.

[34] Penelope Sweetser and Janet Wiles. 2005. Scripting versus emergence: issues for game developers and players in game environment design. *International Journal of Intelligent Games and Simulations* 4, 1 (2005), 1–9.

[35] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N. Yannakakis, and Corrado Grappiolo. 2013. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines* 14, 2 (2013), 245–277. https://doi.org/10.1007/s10710-012-9174-5

[36] Julian Togelius, Jim Whitehead, and Rafael Bidarra. 2011. Guest Editorial: Procedural content generation in games. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (9 2011), 169–171. https://doi.org/10.1109/TCIAIG.2011.2166554

[37] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186. https://doi.org/10.1109/TCIAIG.2011.2148116

[38] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2019. Bootstrapping Conditional GANs for Video Game Level Generation. *arXiv preprint arXiv:1910.01603* (2019).

[39] Michael Toy and Glenn Wichman. 1980. *Rogue.* Game [Atari]. Apyx.

[40] Unity. 2020. *Unity for Games.* Retrieved January 21, 2020 from https://unity.com/solutions/game

[41] Valtchan Valtchanov and Joseph Alexander Brown. 2012. Evolving Dungeon Crawler Levels with Relative Placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering* (Montreal, Quebec, Canada) *(C3S2E '12)*. Association for Computing Machinery, New York, NY, USA, 27–35. https://doi.org/10.1145/2347583.2347587

[42] Roland Van Der Linden, Ricardo Lopes, and Rafael Bidarra. 2014. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 1 (March 2014), 78–89. https://doi.org/10.1109/TCIAIG.2013.2290371

[43] Georgios N Yannakakis and Julian Togelius. 2011. Experience-Driven Procedural Content Generation. *IEEE Transactions on Affective Computing* 2, 3 (July 2011), 147–161. https://doi.org/10.1109/T-AFFC.2011.6

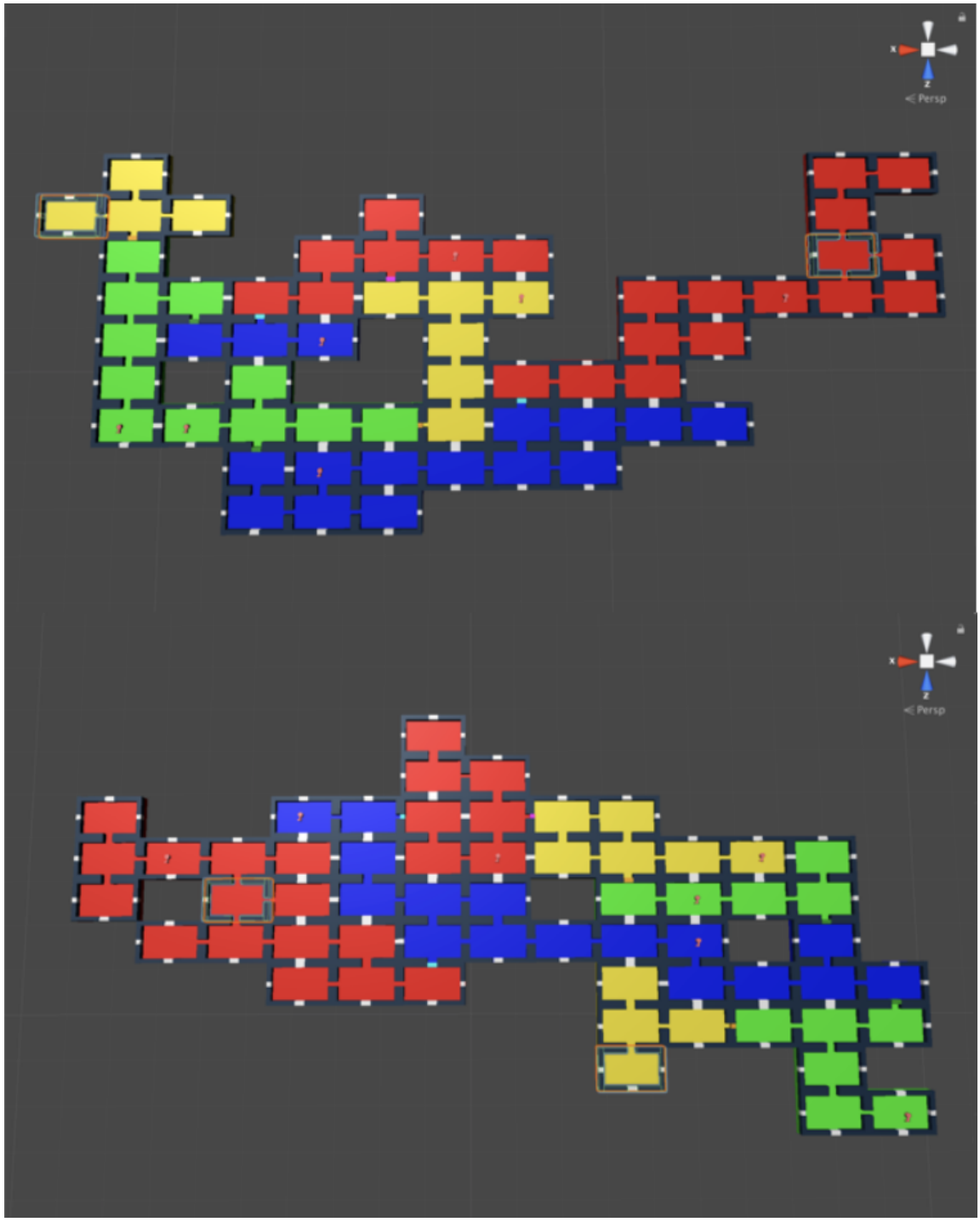

Fig. 6. Samples of the Persistent Subsection Search. Note the increase in branching rooms and clustered sections. Start (red subsection) and End (yellow subsection) rooms highlighted. Subsections differentiated by colours and separated by Locks (coloured doors).

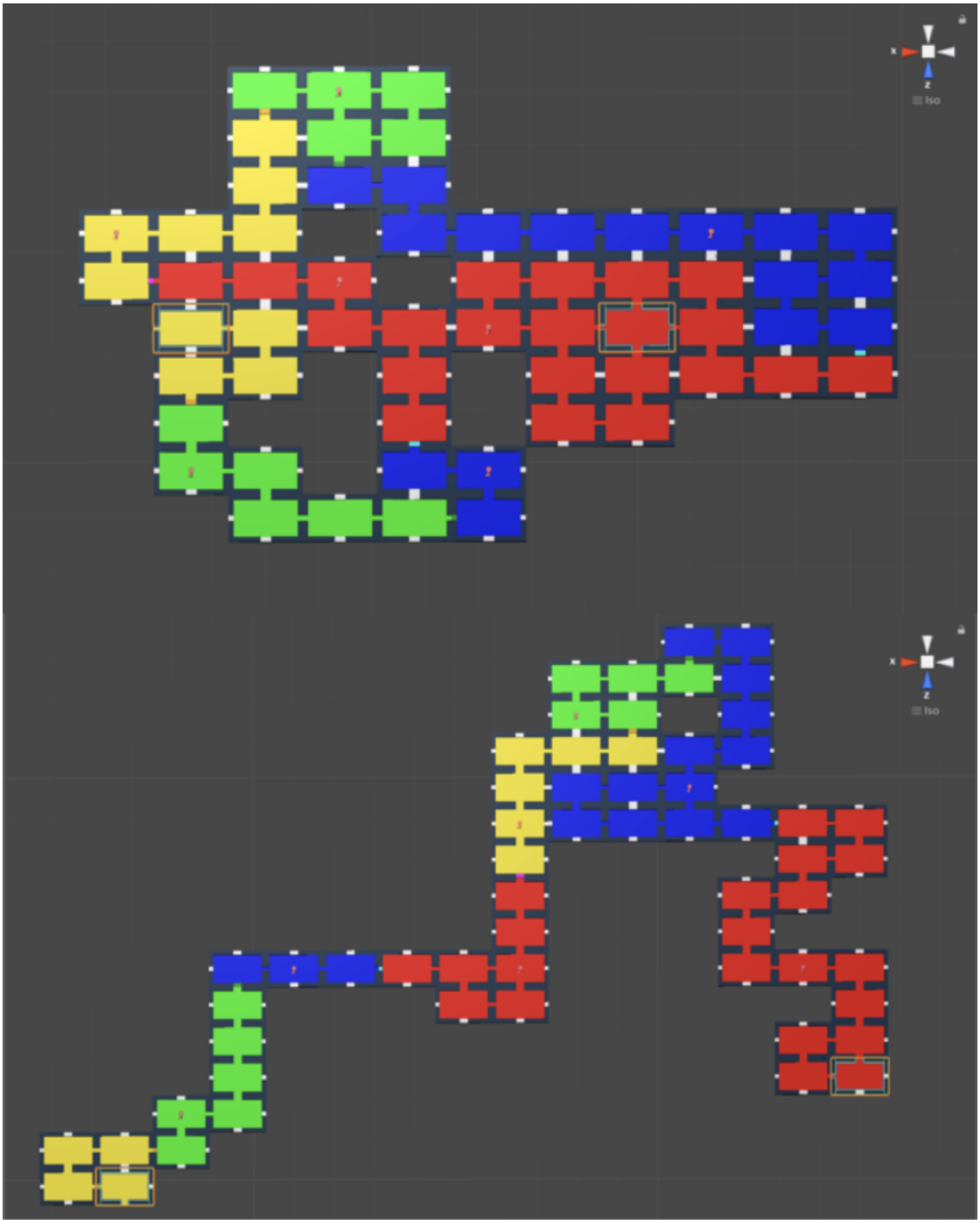

Fig. 7. Samples of the Subsection Search on Halt. Note the long winding corridors. Start (red subsection) and End (yellow subsection) rooms highlighted. Subsections differentiated by colours and separated by Locks (coloured doors).