ICA-4122119

Utrecht University

MASTER THESIS

Graph-defined Dungeons: Exploring Constraint Solving for the Generation of Action-Adventure Levels

Author:
Twan VELDHUIS

Supervisor:
prof. dr. Marc van KREVELD

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Game and Media Technology
Department of Information and Computing Sciences

December 3, 2018

Abstract

Procedural generation is an interesting field to assist the creative process. Recently, constraint solving methods have been getting more focus. An advantage is that they allow for a declarative specification of content. These methods typically restrict themselves to a grid structure. We explore if constraint solving can be applied to dungeon generation and if we can abandon the grid-based nature of these methods. We created a system where designers can define a graph that governs the topology of a dungeon, we then use constraint solving to find pieces that fit this layout. This is done with backtracking and a modified version of Mackworth's AC3 algorithm [1].

Contents

Chapter 1

Introduction

Procedural generation is an interesting topic in various fields, among which game development and architecture. Its primary goal is to speed up development processes by automating various parts of the creative process, as the creation of game assets is an expensive part of game development [2]. In some cases, the generation algorithm is responsible for all of the content in a game. We can typically distinguish two uses of procedural generation: online vs offline generation. Online generation means content is generated on demand when the player needs it. This is typically subject to strict performance requirements. Offline generation means content is generated before the product is released, which means more computation time may be used. However, procedural generation can also be used as an interactive tool for designers, which would be subject to similar performance requirements as an online system. This interactive approach is also called a mixed-initiative generation process [3, Chapter 1].

For the use case of level generation, there are various approaches to tackle the problem. A very simple approach is to create blocks of content with minor randomized parts, which can be linked together. This approach is taken by Spelunky [4] and Ruggnar [5]. Other approaches use grammars to construct levels according to a set of rewriting rules [6], or search for possible levels with a search or evolutionary algorithm [3, Chapter 2].

Recently, constraint solving methods have been getting more focus, with games such as Bad North [7] making use of the WaveFunctionCollapse algorithm [8, 9]. Constraint solving has the advantage that the structure of a level can be defined in a declarative way. With WFC, one would provide an example, and this example serves as a base for the final solution. In Refraction [10], a level is specified with a given mathematical problem that the player needs to solve, in addition to general parameters such as the size of the level, and aesthetic requirements. This, in turn, means it is relatively easy for a designer to get what they want out of a generator, unlike grammars which require careful specification of rules. Constraint solving is, however, typically used on a grid. We think it would be worth exploring how this can be used without using a grid-based system. This could, in turn, create a larger variety in level layouts.

1.1 What are Dungeons?

Dungeons are the typical type of levels in action-adventure games such as The Legend of Zelda series [11]. Dungeons offer a structured level design that can challenge a player in various ways, overcoming the challenges allows a player to progress through the level. The layout of a dungeon is usually a non-linear structure containing rooms and hallways that connect these rooms. Rooms should be interpreted in the abstract sense, they are distinct areas with limited options to pass to other

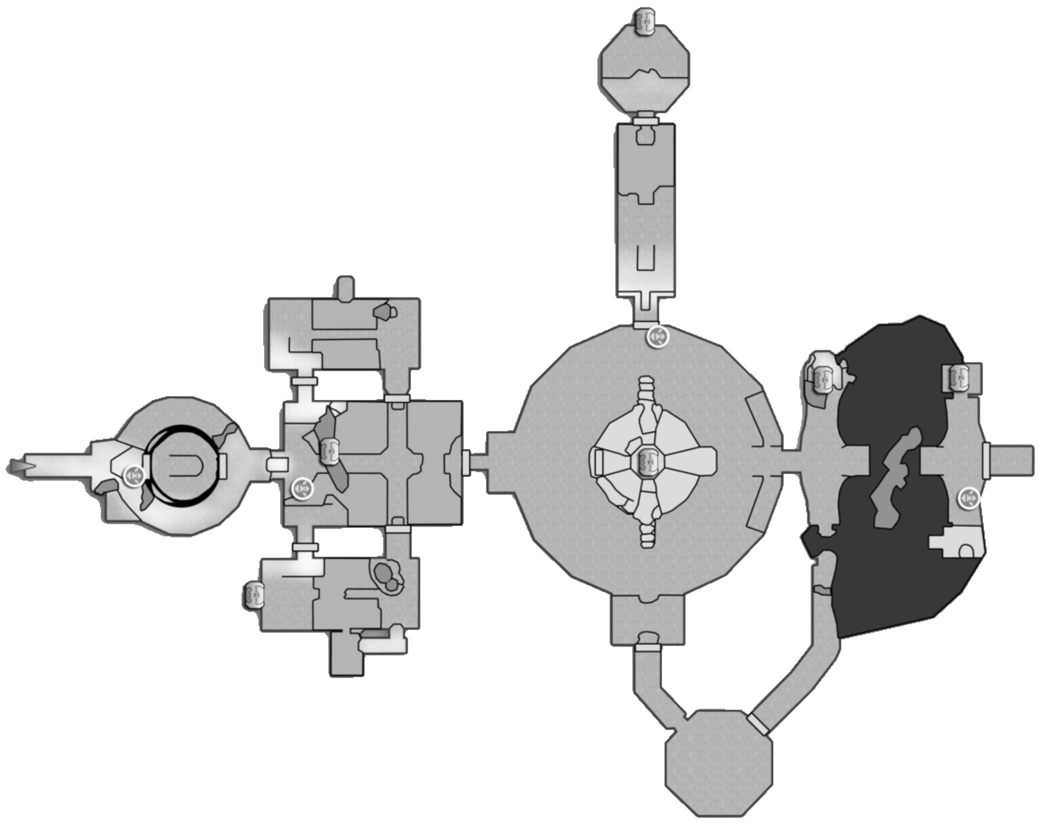


FIGURE 1.1: A map of the Skyview Temple in The Legend of Zelda: Skyward Sword[12]. This image has been adapted from [13].

areas. Hallways are the parts that allow a player to move between rooms. Content in action-adventure games generally fits into one of three categories: Combat, puzzles and dexterity checks. Action-adventure puzzles usually include navigation and lock-key type puzzles. For lock-key type puzzles, a player has to find a key item and use this to get past an obstacle. The puzzle element is that recognizing and remembering which obstacle a key item is useful for might not be trivial. The geometric layout of the dungeon is important for creating these types of puzzles. It typically involves using a non-linear layout, with branches and looping structures. Dormans [6] described a system that can generate dungeons that include these type of puzzles. It uses the concept of a mission, the tasks a player has to perform and in which order, to govern the layout of the dungeon. While it does support branches, it has trouble with loops. Loops are, however, a common occurrence in dungeon design. The typical purpose of a loop is to create a shortcut back to a place the player has been before. This reduces the amount of backtracking a player has to do.

1.2 Outline

In this thesis, we discuss an initial exploration of applying Constraint Solving to dungeon generation. Of particular interest is the promise of using declarative specifications for content generation. In Chapter 2 we discuss work related to the research in this thesis, here we look at various techniques used for dungeon generation, as well as constraint solving in general. In Chapter 3 we consider how we can specify a declarative system for dungeon generation and how this can fit in a larger pipeline, then specify it as a constraint satisfaction problem, and finally discuss how this problem is solved. After that, in Chapter 4, we evaluate the performance of this system and we study the theoretical performance to determine overall scaling characteristics. We also discuss how we set up experiments to determine the performance in practice. The results of these experiments are shown in Chapter 5. Finally, we discuss some observations we made based on those results, discuss future improvements to the system, and describe more extensive evaluation methods.

Chapter 2

Related Work

In this chapter, we discuss several techniques used for level generation that are most relevant for our work. These are constructive methods, grammar-based methods and constraint solving algorithms. While our proposed system is a constraint solver, one can draw parallels to both template-based methods and grammars, and these concepts are essential for how the system fits in the overall pipeline. Other generation techniques are not as relevant for this work, however, interested readers could consult `pcgbook.com` [3] for an overview of procedural generation in general.

2.1 Constructive Methods

In this section, we discuss some traditional level generation methods. Constructive methods are typically fast and simple but could require a lot of manual work.

2.1.1 Space Partitioning

Space partitioning methods generate content by splitting the generation space into smaller pieces. For dungeon generation, these methods have been quite popular, due to the ease of implementation. One example, as provided by Shaker et al. [3, Chapter 3] is binary space partitioning (BSP), in which the space is iteratively cut in two pieces, typically on alternating axes. By randomizing the position of the split plane, variety in size is introduced. In these pieces, a dungeon room can be generated that fits inside the bounds given by the piece. The generation method results in a tree of pieces, which determine which rooms end up being connected. As an alternative, one could choose for a Voronoi diagram, which can have varying results depending on how the points are distributed.

2.1.2 Physics-based Generation

Another way to generate dungeon layouts is by using a physics-based approach [14]. For this, one would first generate a large collection of randomly sized rooms. These are placed at random positions in a small region, such that they all overlap. Then a physics engine is used to push these rooms apart so they no longer overlap. The rooms that are within a certain size range are selected, and the rest is ignored for now. Delaunay triangulation, where each room has one vertex at its center is then used to create a graph from the selected rooms. This graph indicates which rooms are connected to each other. In order to not make the rooms fully connected, the minimum spanning tree is generated from this graph. Several edges from the Delaunay graph are put back, so the dungeon is less linear. Finally, hallways are generated between the rooms that should be connected to each other, non-selected

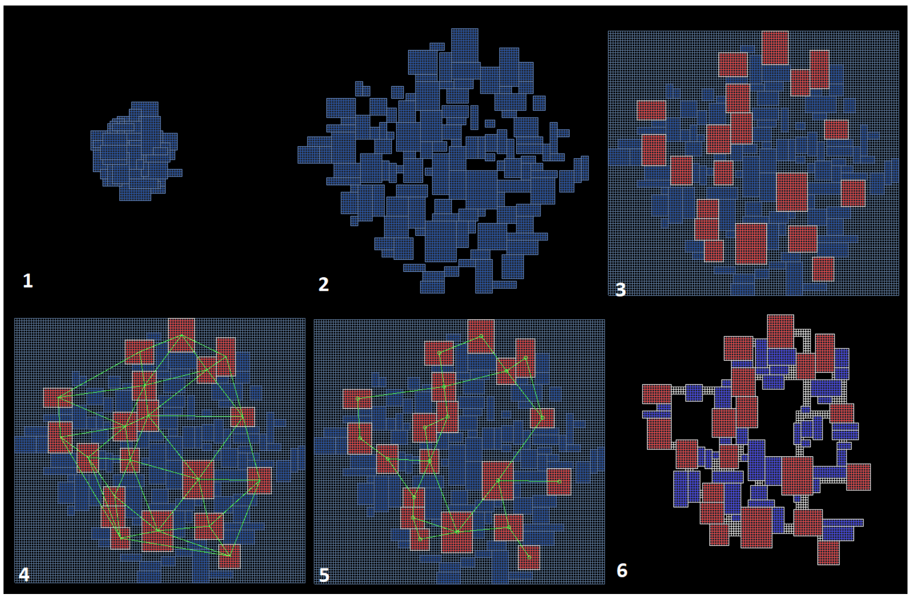


FIGURE 2.1: Physics-based dungeon generation, as explained by [14].

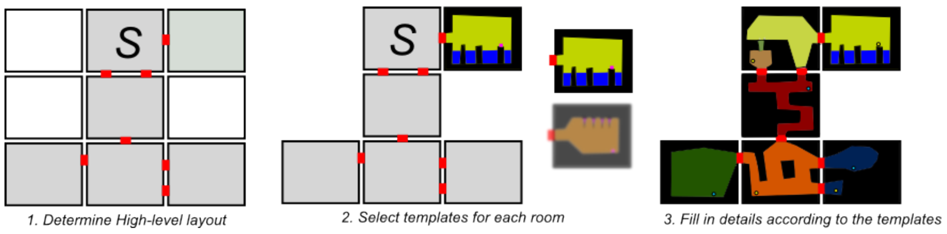

S
1. Determine High-level layout
2. Select templates for each room
3. Fill in details according to the templates

FIGURE 2.2: Generating template-based content similar to Spelunky and Ruggnar

rooms that are on this path are then included in the dungeon, and space on the path not covered by a room is filled in.

2.1.3 Template-based Methods

A simple, but effective generation system is template-based construction. This is used by Spelunky [4, 15] and Ruggnar [5, 16]. The designer first creates a set of templates, these templates are small sections of a level. Then the generator will determine a high-level layout. In the case of Spelunky said layout is generated by a restricted random walk through the grid. Ruggnar takes a different approach in that it first takes out a few tiles, then determines transitions (including a wall) between tiles, and then determines reachable regions with a flood-fill algorithm. It finally selects the largest region to include in the game. In both cases, these result in a few simple constraints on which templates can fit in each tile, and for each tile, a fitting tile is selected with a simple search algorithm. The templates are then finalized with random parameters. Note that the designer should ensure that there is always at least one fitting template in every case. If this is not the case, a more complex system, like constraint solving will be required.

2.2 Grammar-based Generation

2.2.1 Grammars 101

Grammars are a set of rewriting rules. These rules can be used to analyze and describe structures by breaking them down into a derivation tree. This is used in compilers for programming languages to parse a program so it can be translated to machine code or another target language. Grammars can however also be used in the opposite direction, a derivation tree can be generated by starting from a start symbol and iteratively selecting and applying production rules. The resulting tree or string can then represent content.

We consider three types of grammars: text grammars, graph grammars and shape grammars. We also discuss the concept of context-free and context-sensitive grammars.

2.2.1.1 Text Grammars

Text grammars provide the basis of the concept. We can start with a start symbol S, and define several production rules that replace symbols by other symbols. A grammar is simply a set of start symbols and a set of production rules. The original description of these formal grammars [17] also distinguish between terminal and non-terminal symbols. Terminal symbols are symbols that cannot be replaced by a production rule, and non-terminal symbols are symbols that should not exist in the final result. There are, however, grammars that do not use this restriction. An example of a production rule is to replace S by SA, we write this as $S \to SA$. Constantly repeating this rule results in more As being added to the string. All strings that can be generated from this grammar can be considered a language that is described by said grammar. In this case, the language consists of S prepended to any number of As. This kind of rule is a context-free rule, we only rely on a single symbol to replace by something else. We can also create a context-sensitive rule: $xSx \to xyx$. This replaces S by y if it is surrounded by two x symbols. A language that only consists of context-free rules is considered a context-free language, one that also includes context-sensitive rules is a context-sensitive language.

For context-free grammars, it is easy to create a derivation tree, as each newly generated part can still be seen as a separate unit. In Figure 2.3 we visually explain how a derivation tree can be generated with a context-free grammar. We also show that this can be used to represent a dungeon. Alternatively, the generated string can encode instructions to generate content. Prusinkiewicz [18, 19] used this approach to generate vegetation. For this, a special kind of text grammar is used: The L-system. L-systems are categorized by applying production rules in parallel to each matching pattern.

2.2.1.2 Graph Grammars

Text grammars have one major problem in terms of what it can describe, each pattern of symbols is strictly linear. Graphs can describe structures that are more complex than linear (and trees through derivation trees or L-systems) structures, and one can still create rewriting rules to generate graphs. A production rule in a graph grammar matches a subgraph, and then replaces this subgraph by something else.

A problem is that according to Adams [20] using context-free graph grammars is not really a viable option. A context-free graph grammar would limit the subgraphs to be matched to a single node. This implies that edges can not be modified,

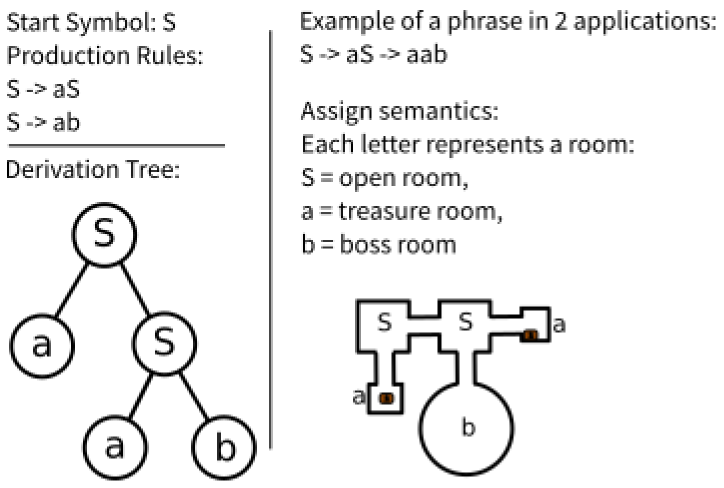


FIGURE 2.3: An example of a derivation tree, how it is derived from a simple grammar and how a dungeon could be generated based on said tree.

this severely limits the kind of graphs that can be generated. Therefore a context-sensitive approach is required.

Unlike text grammars, there is also some variation in how rules and the application of them can be defined. One approach splits production rules from connection rules. This first removes connections involving the nodes that a production rule is applied to, then applies the production rule, replacing nodes, and then reconnects nodes according to the connection rules. This can somewhat reflect a context-free system, however, the trade-off is that connections are not preserved, even if only a single node is replaced. Another system only has production rules, and this method preserves connections. This requires adding identifiers to each node in the production nodes, so the algorithm can match nodes from the left-hand side to the right-hand side of the rule.

2.2.1.3 Shape Grammars

Even more complex are shape grammars. Instead of using symbols, one can use shapes as the units to replace. In a pure shape grammar, shapes can even be generated through emergence. Shapes that are not the right-hand side of a rule and do not exist initially, can in certain situations still get generated. These new shapes might still be matched in the left-hand side of a rule. Doing this is hard as it requires spatial pattern matching. A more simplified model just considers objects, which may be scaled, rotated and translated to generate more complex shapes. This is the context-free interpretation of shape grammars and can be very effective to generate geometry.

2.2.2 Dungeon Generation with Grammars

Dungeon Generation with grammars has been getting research attention for quite a while. In 2002, Adams [20] proposed a system that generates a dungeon layout from graph grammars, and then create missions inside this dungeon. Missions are graphs that describe the way a player has to traverse through a dungeon. Missions are then used to place key items. In 2010, Dormans [6] proposed a similar system, that starts with generating the mission and creates dungeons based on the specified mission.

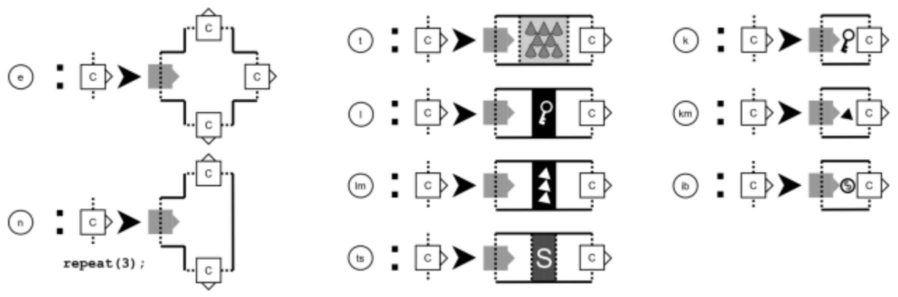


FIGURE 2.4: An example provided by Dormans [6] that specifies shape grammar rules depending on mission graph nodes

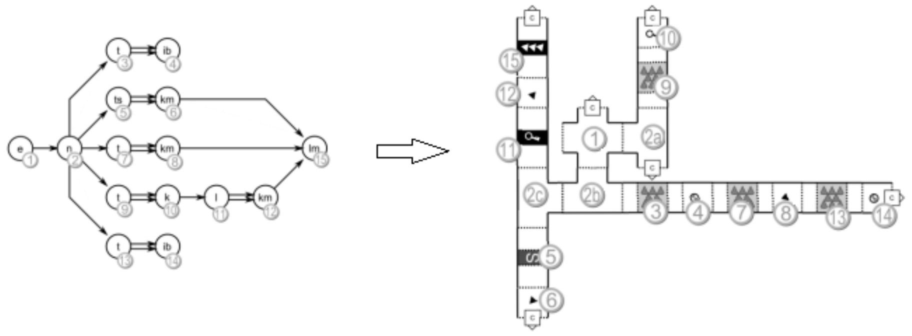


FIGURE 2.5: An example of how a mission is used to create a dungeon. The mission graph on the left is drives the selection of production rules as specified in Figure 2.4. Together they form a grammar that results in the dungeon on the right. The image was provided by Dormans [6].

We visualize said system in Figures 2.4 and 2.5. This system was later extended to include an intermediate space graph, allowing for content not directly related to the mission. Van der Linden [21] described a method to define grammars for generating missions based on the game's mechanics.

2.2.3 Grammar Networks

In an attempt to improve the level of control for designers, Middag designed a grammar-based system called Designer Controlled Grammar Networks [22]. This system brings a few advances to the approach used by Dormans:

- Graph nodes may hold attributes, and production rules can work on just a set of attributes.
- The left-hand side of production rules can have programmable conditions, and is not limited to simply matching subgraphs and attributes.
- Rule selection, which is usually based on the attributes in the mission graph, can be completely programmed.
- Constraints may be applied to grammars, which can be fixed using special production rules.
- Multiple grammars and their respective structures may interact, in a complex designer specified network. This means that, for example, a production rule

for the grammar that generates a mission graph can depend on a node in a space graph.

A major disadvantage, however, is that the system relies on a web of cyclical references, meaning that while control is available, it is not necessarily intuitive.

2.3 Constraint solving

Constraint solving is gaining more ground in recent generation methods, providing effective control for designers while still allowing for a good variety in generated content. It helps with generating exactly the kind of content a designer wants. In the following three sections, we discuss various constraint solving methods and work that explores or applies these methods.

2.3.1 Finite Domain Constraint Solving and the AC3 algorithm

The basic form of constraint solving is solving finite domain Constraint Satisfaction Problems (CSPs). Here we have a set of variables and relations between variables. These variables can have a finite number of values. The CPS then defines a finite set of constraints, which can fit into three categories:

- Comparative or relational constraint: Requires a specific relation between two variables to be satisfied.
- Cardinality constraint: Limits the number of variables that can take on a certain value. This is a global constraint that can affect all variables.
- Narrowing or domain constraint: Limits the values a variable can take on.

Foged and Horswill [23] explained well how the solver can be optimized with several steps, starting with a brute force approach, and finishing with an optimized algorithm with the following features:

- Backwards checking: Avoid continuing building a solution when a constraint is already violated.
- Constraint propagation: Avoid attempting solutions that can be deduced to fail given the current state. This is done by deriving additional narrowing constraints as the search space is explored.
- Backtracking: In case a failing solution is found, the problem state is rolled back so that another path can be attempted without restarting the algorithm.

In particular, such an optimized algorithm is the AC3 algorithm (Arc Consistency Algorithm #3) [1]. While even more optimized algorithms exist, this algorithm is relatively easy to implement and therefore quite popular. This algorithm is specifically about the method of constraint propagation and mainly considers comparative constraints.

The algorithm considers the CSP as a graph, called a constraint network. Each node represents a variable, which can have a domain of possible values. The CSP provides a set of narrowing constraints. The domains for each node are adjusted to be consistent for these narrowing constraints in the initialization process. Then, the CSP also provides a set of relational constraints or comparative constraints. These are considered edges in the graph. The solver relies on choosing values from the domain for a single variable each iteration. This choice is then propagated. AC3 keeps a set of nodes Q that have new information, which starts with the chosen variable. It then iteratively removes one node from Q and checks all values for consistency in their neighbors. Each neighbor that had its domain adjusted gets added to Q. The algorithm continues until Q is empty, or a domain becomes empty.

2.3.2 Answer Set Programming

ASP is a method of defining and solving CSPs without having to create a custom solver algorithm. One would write the CSP in a domain specific language, after which the problem is solved with generic solver tools such as Clingo [24]. These tools, in turn, provide optimized implementations of constraint solving algorithms. Programming the CSP is typically done in AnsProlog, a language based on predicate logic. AnsProlog has two main constructs: Facts and Rules. A fact is an expression that is considered true. Rules can be used to generate or deduce new facts. There are three different types of rules:

- A **choice** rule describes how a new fact may be guessed.
- A **deduction** rule describes how new faces can be derived from existing facts.
- An **integrity** rule describes that a certain combination of facts are forbidden in a valid solution.

ASP allows for a large variety of constraint problems, however, it requires understanding a relatively complex logical language. Aside from that, existing solvers can often not work directly with the level representation, so a translation algorithm is required to integrate ASP into the game or development pipeline. If this is a complex task or results in undesired overhead, it could be a benefit to use a more domain-specific solver.

2.3.2.1 Application

ASP has been explored for puzzle generation in Refraction [10]. Refraction is a game in which lasers must be redirected, split and combined in such a way that a set of receivers receive the correct power of laser light. The purpose of this game is to teach players mathematical concepts of fractions. Puzzles are created by creating a conflict between the lasers' output and the receiver's required input, as well as by obstructing a straight path between laser and receiver. This second aspect means that the layout of the level is important for determining the difficulty of the puzzle.

Smith et. al. set up several requirements for the generation system:

- A level should obviously be solvable.
- Tweakable difficulty in terms of logical and mathematical skills required to solve a puzzle. This means that alternative solutions can only be allowed if they also meet these requirements.
- Aesthetic requirements, including size, distribution of game elements and specific arrangement of elements.

The problem treats all those requirements as equal. A level that doesn't meet the aesthetic requirements will be rejected in the same way that a level that is not solvable is rejected; they are all hard constraints.

Smith et. al. compared the performance of ASP and a constructive method they created specifically for this game. We first discuss the general approach to the problem, which is shared between the two methods. Similar to Dormans' approach to split dungeon design in missions and spaces, the puzzle generation problem is also split into missions and spaces. A mission for Refraction describes the topology of the laser system as a Directed Acyclic Graph (DAG), and thus a primary solution independent of its spacial location. The input of the mission generation system is expressed as a mathematical expression which the player needs to solve the puzzle. Several other parameters are given for use of the spatial part of the puzzle. Spaces are generated by a concept they call grid embedding. Here the mission is embedded in the grid, in such a way that the solution specified in the mission DAG is possible. Additionally, requirements for the number of blocking elements and aesthetic requirements must also be resolved. Finally, after grid embedding is completed, the resulting puzzle is solved to search for alternative solutions, which are then additionally tested against the constraints.

The constructive method starts with the mathematical expression, derives an expression tree from that and transforms that in several steps into the mission DAG. The search method then embeds this DAG into the grid and solves puzzles. ASP is capable of dealing with the entire process, for details on which rules are specified, we refer to the paper by Smith [25].

In their analysis that compares the methods, they noticed that ASP takes significantly less code. Performance-wise, mission construction has a similar execution between the constructive and ASP method, although ASP has some initialization overhead, making constructive faster for single generation runs. Grid embedding is about 5 times faster with ASP compared to search-based on average, but also here overhead is a thing. Puzzle solving is very slow with the search method, while ASP can solve it quickly. Qualitatively, ASP is powerful as it allows for quick iteration on the generation constraints. Additional requirements can be added without having to rewrite the design of the algorithm.

2.3.3 WaveFunctionCollapse

WFC [8, 9] is a constraint solver designed to create textures locally similar to example textures. It is inspired by discrete model synthesis. Discrete Model Synthesis was first described by Merrell [26]. He formally described the problem of creating new models out of examples. For discrete model synthesis, the models consist of a grid of labeled geometry. The model can, therefore, be reduced to a labeled grid, a texture with a finite number of values per pixel. To create new models from example models, he defined the adjacency constraint, two labels may only be adjacent to each other in the generated model if they are also adjacent in the example model. He introduced an algorithm based on constraint propagation that solves this problem.

WFC extends on this by adding support for larger patterns, instead of just adjacency constraints. The algorithm works in two steps: Pattern extraction and the actual constraint solving step. Patterns are extracted from an example model and placed into an index containing each pattern, with optionally rotated and mirrored versions. The constraint solving algorithm attempts to solve two constraints to achieve local similarity:

- C0: Each pattern should occur at least once in the example model.
- Weak C1: The distribution of patterns in the output and the example model should be similar. Note: Hard C1 would mean exactly equal distributions and this would contradict C0 in some cases.

Inspired by quantum mechanics, it takes some concepts as metaphors. The to-be-generated model is called the wave (function). Each state (pixel) on the wave is initialized in a superposition of all possible patterns. This is similar to the set of possible states mentioned in Section 2.3.1. This superposition is represented as a boolean array at each pixel, with each element of the array representing a label, and the value specifying whether it is an allowed value. The algorithm then iterates through two phases:

1. Observation, which consists of selecting a state and choosing a new state.
2. Constraint propagation.

State selection is done based on the minimum non-zero entropy heuristic. Entropy is defined as in the following equation:

$$Entropy(p) = \begin{cases} \text{undefined} & \textbf{if } sum(p) = 0 \\ 0 & \textbf{if } sum(p) = 1 \\ sum(f * p) + \epsilon & \textbf{otherwise} \end{cases} \tag{2.1}$$

In this equation, p is the boolean array (true = 1), f is an array of frequencies for each pattern in the example model, and ϵ is a small random number that serves as a tiebreaker. If entropy is undefined, it means a contradiction has been found. The algorithm aborts in this case, as it does not implement backtracking. In practice, a contradiction is rare, as stated by Gumin [9].

Finally, note that the wave state is based on patterns, not label values. This means that the wave does not represent the final model when all values are collapsed, the model can be extracted based on the assigned patterns. Gumin did implement a way to visualize intermediate solutions.

It is also possible to constrain specific values before and while running the algorithm, which in turn reduce the number of possible patterns at that pixel. This allows for a mixed-initiative approach, as well as interaction with other generation algorithms. Stalberg [27] created a web-application showcasing how it can be used in a mixed-initiative manner.

WFC has been used for generation of levels in various games such as Bad North, Proc Skater, and Caves of Qud [7, 28–30].

Chapter 3

Methods

As mentioned in the introduction, we want to create a declarative way to define the structure of a level, and let a generation system create content that fits said structure. We focus on a constraint solving system to do this. Similar to the approach taken for Spelunky and Ruggnar, as explained in Chapter 2.1.3, we allow a designer to create several modules that can be used in the generation process.

These modules consist of three parts: geometry, a skeleton and collision information. The geometry is, in our implementation, a simple game object, and can include actual game functionality. Collision information can be optionally included with the geometry and specifies the bounds of the module. Finally, skeletons specify how modules can connect to each other through *connectors*. Each connector specifies a plug type, which drives constraints on which modules can connect with each other, and/or through which connectors. Plug types can represent the type of transition between modules. These transition types can represent hallways, tubes, vertical drops or even multiple hallways. Unlike the Spelunky-method and many constraint solving methods, these modules don't need to be on a grid. We also don't need a full coverage of modules to generate content. Removing this grid restriction means several optimizations can no longer be used in the constraint solving system. One problem this introduces is that one can no longer guarantee that modules perfectly fit. Thus, one would have to test whether modules can be made to fit into the system. We will describe this process in more detail in Section 3.3 and 3.4. First, we look into how our system can fit in a larger pipeline, then we specify our input. We then specify the constraint problem in a more formal way and finally describe the way our system solves it.

3.1 The Generation Pipeline

As mentioned before, the proposed pipeline uses a set of small chunks of a level we call modules. Multiple levels can be generated using those modules, based on a graph that specifies topological constraints on the level. The graph can also specify additional constraints like the set of modules allowed on a given node, and how they need to be connected. The pipeline consists of the following four steps:

1. Design or generate modules, small chunks of a dungeon or level.
2. Design or generate a graph that specifies the topological layout of the level.
3. Run the constraint solver to assign modules to the nodes of the graph and place them in the game world. The output of this step is the full geometry of the dungeon.
4. Run scripts on each module to decorate the dungeon and to configure game mechanics.

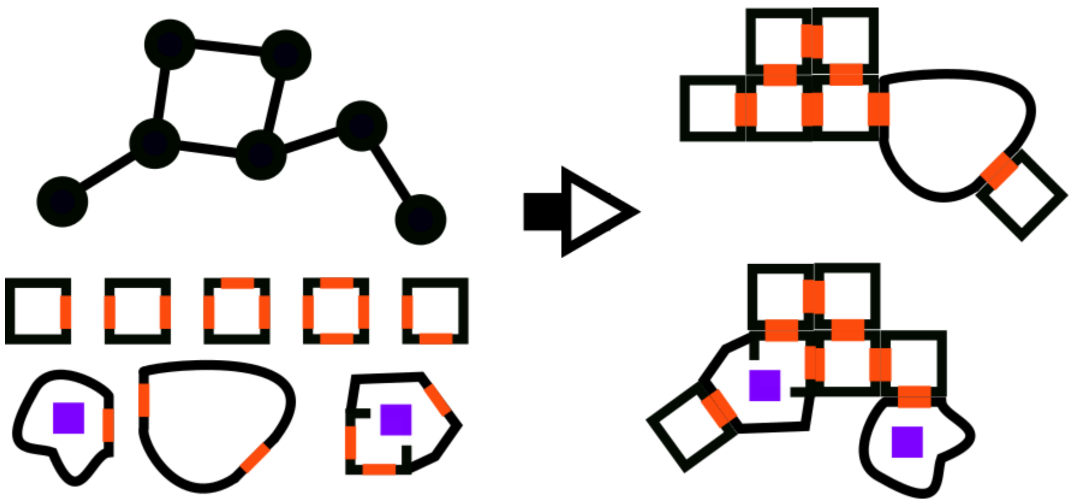


FIGURE 3.1: A schematic visualization of the input and output of the system. It takes a graph and a set of modules and outputs a dungeon made from the given modules, in a way it fits the graph.

This pipeline allows for both human and automated design of levels, while the overall geometry is generated with constraint solving. From these specifications, we can derive the requirements for our constraint solver.

First of all, we take as input a graph and a set of modules. The goal is to assign each node in the graph a module and in a way that these modules can seamlessly connect to each other. More specifically, the modules should connect as specified in the graph. Additionally, modules cannot overlap.

3.2 Input

3.2.1 Modules

A module is a game object consisting of three components: A skeleton, a collider, and the geometry. The latter may include other colliders and implement game mechanics. For our purpose, we consider the geometry a black box that is carried with the module. The skeleton defines where, and in which direction, other modules can be attached, by specifying connectors. The collider specifies the bounds of the module, to capture the constraint of no overlapping modules. For user convenience, we do provide the option to specify the collider with the geometry itself.

Ideally, we want to support flexible modules, rather than purely rigid components, however, this does make the constraint problem more complicated. We will define an interface that may support flexible modules, however the implementation is left for future research. To make modules more flexible, we need to open the black box of geometry a bit, and link parts of the geometry to connectors. We also need to specify the range of freedom that each connector has, both rotationally and in a prismatic sense. Additionally, while connectors can be represented with a local coordinate system (position and direction) in the rigid case, to support rotations and prismatic translations, we need to define the connector as an arm. This allows the connecting position to be in a different location from the rotational pivot. We call the position of the end of this arm the plug position.

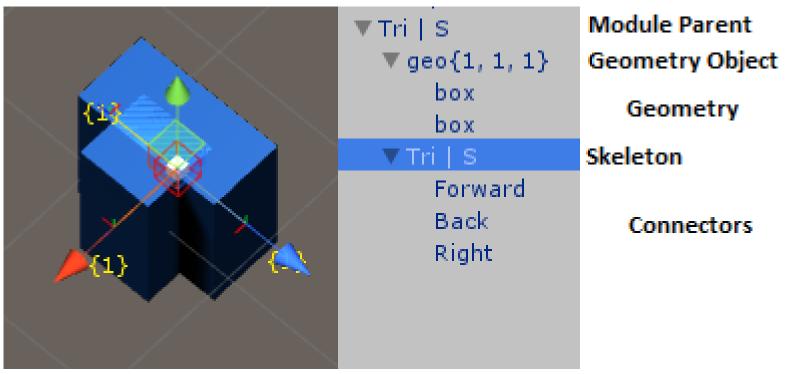


FIGURE 3.2: Implementation overview of the module specification. We keep the skeleton part separate from the geometry. Various components are set to the correct values automatically using code that executes in edit mode. The name of the skeleton is set to list all labels on the module, separated by the pipe character (|).

3.2.2 Plug Types

Plug types define the type of connection between two modules. This can represent variations in hallways in terms of shape, size, and decoration. However, they can also capture other concepts such as a drop in elevation. The constraint solver has to make sure that plug types match. However, in order to support one-way passages, such as a drop in elevation, it is possible to have two variants or genders of the same plug type. If two genders are defined, we specify that connecting modules must use different genders of the plug to properly connect. Plug types that support this asymmetric relation may only support asymmetric connections.

Plug types are specified with the module and assigned to connectors.

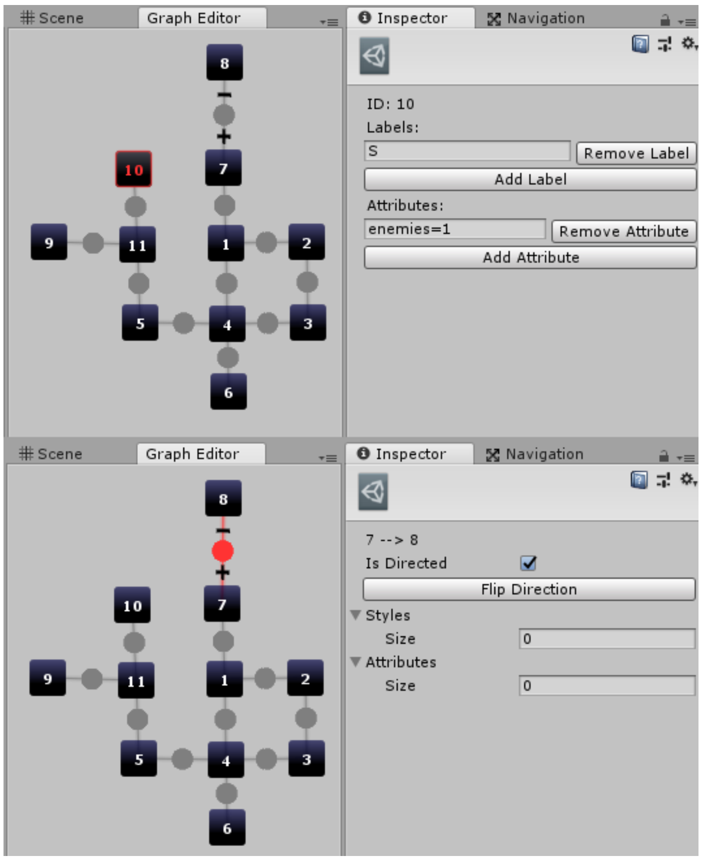
3.2.3 The Level Graph

The level graph describes the topology of a level and describes module set constraints on nodes by means of labels, and plug constraints on edges. Plug constraints cover both the plug types available for an edge, and optionally also which node uses which gender, making the edge asymmetric. In addition to these constraints, we allow users to specify attributes, which can be used in a further pass as gameplay information.

We implemented a graph editor in Unity, which is displayed in Figure 3.3. This editor allows the user to create new nodes, edges between nodes and edit constraints and attributes. Due to this freedom, it is, however, possible that we don't have a connected graph. To deal with this, we can delete lone nodes and if multiple connected subgraphs exist, we remove the smallest ones. This is a manual pruning option for the user. When generating, this pruning process is applied to a copy of the graph, which is then used in the generation process.

3.3 Formalizing the Constraint Problem

As input, we have a module set M_{all}. Each module m in this set has a set of labels, which we denote $L(m)$. We also have a graph, where for each node i, there is a variable constraint which consists of one or more module labels (if none are specified,



ID: 10
Labels:
S | Remove Label
Add Label
Attributes:
enemies=1 | Remove Attribute
Add Attribute

7 --> 8
Is Directed ✓
Flip Direction
Styles
Size 0
Attributes
Size 0

FIGURE 3.3: The graph editor we made in Unity to specify our level graphs. In the implementation the term "styles" is used for "plug types".

we default to its complement, thus all possible labels). We write $S(i)$ for the set of possible labels that are consistent with the node.

The set of possible modules on node i, denoted $M(i)$ can be defined by the following equation:

$$M(i) = \{m \in M_{all} | \exists s \in S(i) : s \in L(m)\} \tag{3.1}$$

In other words, the set of possible modules on node i is any module that has a consistent label for this node. This constraint is resolved in the initialization phase of the algorithm.

The next constraint to consider is the plug constraint. Each edge between nodes needs to have a consistent plug type, and in case of asymmetric plugs, they need to have opposite genders as well. For this, we consider the concept of an arrangement. Since each module has a fixed set of connectors, with a fixed plug type (and gender), we can say that there are a fixed number of ways a module can be connected to their neighbors. We, therefore, match up each individual connector with a neighbor. We call this an assignment, which also carries information about the plug type and gender. Assignments have four properties: neighbor $n(a)$, plug type $p(a)$, plug gender $p'(a)$ and its connector $c(a)$. An arrangement is identified by a module identifier combined with all assignments. When initializing the problem, we, therefore, need to find the domain of arrangements; this is done by determining every permutation of neighbors. We can then narrow this set based on plug consistency. The goal of the algorithm is to find an arrangement for each node in the graph.

To assist with this problem, we also include the domain of plug types in our system. We write $P(e)$ for a domain of plug types on an edge e. $e = (i, j)$ is an un-ordered pair between nodes i and j. An arrangement $A(i)$ on node i is plug consistent if the following predicate holds:

$$\forall a \in A(i) : \forall a' \in A(n(a)) : p(a) = p(a') \land p'(a) = -p'(a') \land p(a) \in P((i, n(a))) \tag{3.2}$$

In this equation, $n(a)$ is the node referenced by assignment a (a neighbor of i). $p(a)$ is the plug type of a, and $p'(a)$ is the gender of said plug. In this case, we encode two genders with 1 and -1, where a symmetric plug will have a gender of 0. The user may specify a narrowed version of $P(e)$ in the level graph. By default, it will consist of every plug type used in the module set.

We further specify that a plug type has to be either symmetric or asymmetric, but cannot support both at the same time.

$$\neg\exists\ a_1, a_2 \in \{a \in A(i) | i \in G\} : p(a_1) = p(a_2) \land |p'(a_1)| \neq |p'(a_2)| \land a_1 \neq a_2 \tag{3.3}$$

Additionally, we don't support dangling connectors; all connectors in a module must connect to a neighboring node.

Finally, the user may specify gender constraints. Gender constraints can specify that an edge must have an asymmetric relation (given edge (i, j), then for all assignments in all arrangements on node i: $n(a) = j \to p'(a) \neq 0$ and similarly for node j).

3.4 Solving the Problem

Now with a formal definition of the problem in place, we can discuss our approach to solve this problem. We created our solver in C#, and a large part of the solver can be used outside a Unity environment with a small amount of additional work.

3.4.1 Overview

Constraint solving is an iterative process, in which a decision for a variable is made each iteration, checked for consistency and then propagated through the problem. This propagation system then allows for reducing the number of decisions to make. If an inconsistency is detected during this process, the system can then backtrack to try other choices. Our system is based on the same concept but has some subtleties that need to be dealt with.

We first provide a description of the overall system. This system is also visualized in Figure 3.4.

First, we initialize the problem state, after that, we start an iterative process with a given maximum number of iterations. Each iteration, we start by checking if we found our solution already, in which case the system terminates. This is the case if all nodes are verified, and thus have a domain containing only a single arrangement. If we did not find a solution yet, we choose a node to make an arrangement decision for. After that, we decide on an arrangement A for this node. This arrangement is then tested; we check if it fits and does not violate any collision constraints. If this test passes, we first create a backup of our problem state, including the decision, and then we set the domain of the node to only contain arrangement A. We can then consider the node validated. If the test did not pass, we remove arrangement A

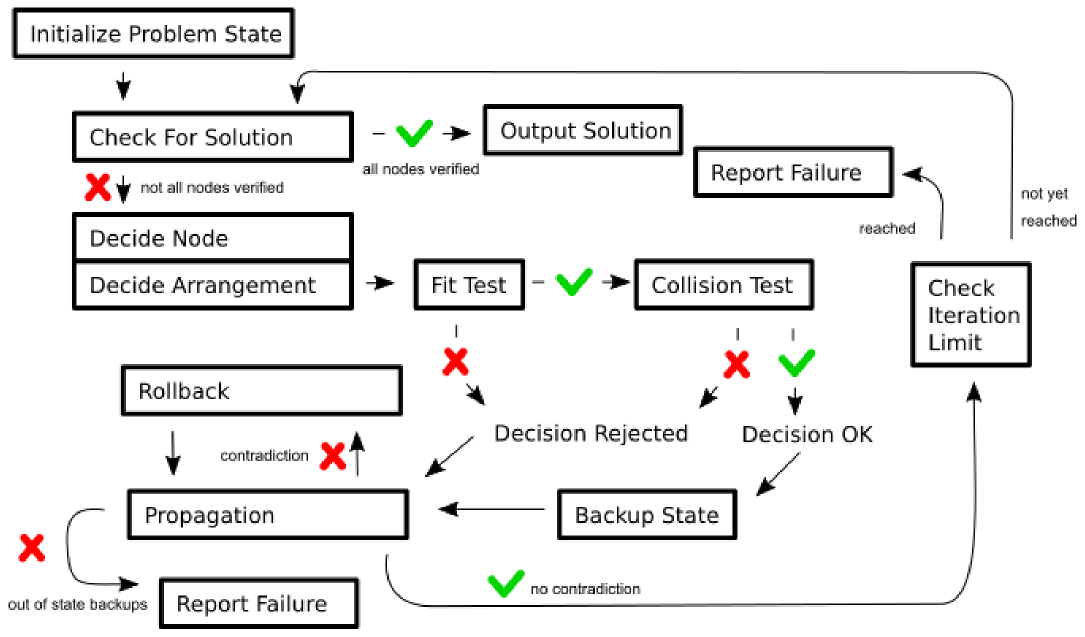


FIGURE 3.4: A schematic overview of the system.

from the domain of the node. After either case is handled, we proceed to propagate this modified domain through the problem state. When propagation does not result in a contradiction, we continue with the next iteration. If it, however, does result in a contradiction, we perform a rollback to the previous state. We then remove the coupled decision from the state and propagate again. The system will continue performing rollbacks and propagation until either we find no contradiction anymore, in which case we can continue to the next iteration, or run out of backup states. In the latter case, we report there is no solution to the problem and terminate.

In the next few sections, we explain the problem state, decision model, verification system and propagation system in more detail.

3.4.2 Problem State

As mentioned before, we start with a set of modules and a graph, where nodes may be labeled and edges may be constrained to certain plug types. We model our problem based on a possibility space of arrangements for each node, and a possibility space of plug types for each edge. To simplify the representation, all modules, labels and plug types are represented by integers. An array is created that allows for a mapping between these integers and the actual objects.

For each node, we then store the set of possible arrangements (domain of arrangements) and a boolean variable that indicates whether the node is verified. We also store a set of possible plug types for each edge. These sets are put in a class that provides a copy functionality, and instances are placed in a dictionary, allowing fast access through node IDs and un-ordered pairs of node IDs that represent edges.

In order to ensure consistency of the symmetry of plug types, we also store a list of asymmetric plug types.

The resulting state can be described with (V, A, P) as the mutable part, where V and A are accessible by nodes, and P accessible by edges. The asymmetric plug list and the graph is the immutable part of the state.

3.4.2.1 Initialization

To perform initialization of the state, we start by transforming all modules, plug types and the nodes themselves in an integer representation. After that we build the asymmetric plug type set. We then look for plug type inconsistencies in the module sets and throw an error if any occurs. That is, if in a module set, a plug type is used as both symmetric and asymmetric, we can't properly infer what it should be.

After that, we initialize A by first building the set of possible modules for each node, which is defined by Equation 3.1, which is then pruned by the number of edges. We then create permutations of connectors and their neighbors, creating assignments. Each module is then combined with each permutation to create the domain of arrangements.

P is generated by first setting it to the set of all plug types for all edges. If edge constraints are specified, we set it to the constrained set instead (it lists all possibilities at this point). V is simply initialized with all nodes being unverified.

Finally, we narrow the edge information, for each edge based on the existing arrangements, similar to the `validPlugs` function, which will be discussed in Section 3.4.5.

3.4.3 Decision Model

When determining which node to decide an arrangement for in an iteration, we first need to determine which are possible candidates. While in a typical constraint problem, one can pick any variable, we need to deal with the issue of fitting modules together. This will be explained further in Section 3.4.4; however, the main concept is that we cannot pick a node that is not adjacent to an already instantiated node. If we did pick a node that is not connected to something already instantiated, we cannot determine whether the module actually fits, nor do we know where to instantiate said module. Because of this, we need to keep track of which nodes are able to be chosen and update this every iteration.

This frontier of possible choices is the full set of nodes in the graph if there is no module instantiated yet (or no node is validated), and consists of all unverified nodes with a verified neighbor.

With a set of possible options available, we can now choose a node. The simplest method would be to choose randomly, however, we chose for the same heuristic used in WaveFunctionCollapse [8], namely the minimum entropy heuristic. Entropy is defined by the size of the domain of arrangements (minus one) on a given node and a small random offset $-0.1 \leq \epsilon \leq 0.1$ to deal with ties. Unlike WFC, we pick the actual minimum entropy, rather than the minimum non-zero entropy, since we want to choose nodes with only one valid arrangement in order to validate them. Subtracting one is, therefore, not required. In Figure 3.5 we visualize an example of this process.

Once a node has been selected, we randomly select an arrangement.

3.4.4 Verification System

An arrangement that has been decided on needs to be verified. To do this, we, first of all, need to instantiate a module. This requires access to the node ID, a mapping from node ID to already instantiated modules and the arrangement itself, which provides info about the relevant neighbors and which module is picked.

Instantiation then starts by looking for instantiated neighbors. Here we consider three cases: There are no module instances yet, there is one instantiated neighbor,

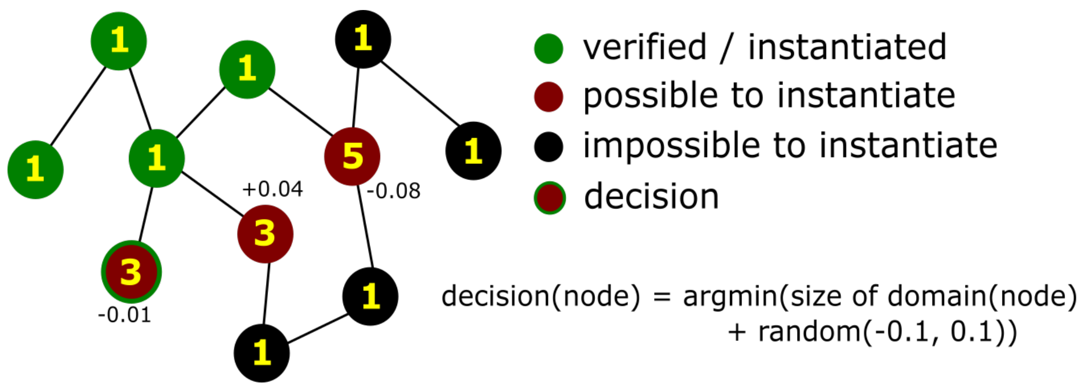


FIGURE 3.5: A visualization of the decision model. In an arbitrary iteration, the problem state might look as shown in the image. The green nodes already have an instantiated module. We consider any node with an instantiated neighbor, which are the red nodes. We then evaluate the entropy of these nodes to decide the next node.

and there are multiple instantiated neighbors. In the first case, we can just create an instance of the module, place it at the origin with default rotation, and consider it verified. This case would happen on the first iteration in the algorithm. For the second case, we instantiate a module and align it so it fits with this neighbor. We then verify collisions. In the last case, we choose an arbitrary neighbor (whatever pops up first) and instantiate a module to align with that module. We then check for the remaining instantiated neighbors if it fits; this check allows for a certain error the user can specify.

3.4.4.1 Instantiation

When instantiating modules based on a given neighbor module, we need to determine the position and rotation of the new module. Here we consider that two connectors need to form a straight line and connect with a matching local up-vector. We can, therefore, align the two connectors and then transform this alignment back to the module itself.

We start by instantiating the new module at the origin with default rotation. After determining with which connectors the new module and its neighbor should connect (C_{new} and C_{other}), we look at what the correct rotation of the module should be. C_{new} has a different rotation and plug position than the module, so we first determine the rotation required to align it with C_{other}. For this purpose we use the forward vectors (fw) of both connectors, and determine the angle from $fw(C_{new})$ to $-fw(C_{other})$ rotated along the up-vector of C_{other}. Notably, we use $-fw(C_{other})$ as we determine the rotation required such that both vectors are equal, while the target is to get the vectors point in opposite directions. We then convert this rotation to a quaternion representation.

$$\texttt{R = axis\_angle(signed\_angle(fw(}C_{new}\texttt{), -fw(}C_{other}\texttt{), up(}C_{other}\texttt{)), up(}C_{other}\texttt{))} \tag{3.4}$$

We then determine the position of the module by taking the plug position of C_{other} and from that subtract the plug position of C_{new} rotated by the rotation we just determined. This is valid because the alignment requirement demands that the target position of the module, plus the rotated position of C_{new} should end up at the same position as C_{other}.

$$target + R * pos(C_{new}) = pos(C_{other}) \tag{3.5}$$

$$target = pos(C_{other}) - R * pos(C_{new}) \tag{3.6}$$

3.4.4.2 Collision Checking

In our module specification in Unity, we allow users to specify box and sphere colliders. A Unity script then creates a compound collider from these colliders. We use a custom implementation of collision checking in order to avoid using Unity's physics engine and keep full control over the execution of the algorithm. We also keep full control over which objects are valid for checks. This makes it easier to include the system in a project since we don't need to deal with collision layers. An additional advantage is that porting the system out of Unity would be easier.

A module is checked for collision with any module except their direct (instantiated) neighbors. If any collision occurs, we consider the arrangement of the module invalid.

3.4.4.3 Instance management

As the problem is being solved, we instantiate modules for certain nodes. We need to be able to access these modules in order to instantiate and validate other modules. We also need to be able to clean up instances in case of a rollback. For this, we keep a dictionary of all instances that maps from node-ID to a tuple that specifies the module instance and which arrangement is specified with that instance. When a rollback happens, we check for each node if it is still considered validated in the problem state. If it is not validated, we delete the instance.

3.4.5 Constraint Propagation System

The most important part of the algorithm is the constraint propagation system. This makes the solving process reasonably efficient. When a decision has been verified, whether it is successful or not, or we just performed a rollback, we propagate new information about the domain of arrangements on a node to other nodes in the graph. We also update the domain of plug types on edges between those nodes. The system is very similar to the AC3 algorithm discussed in Section 2.3.1.

We show this process with the following pseudocode:

```
Data: changedNode, graph, V, A, P
Result: Whether propagation was successful without contradiction
S ← new stack
push changedNode on S
while S not empty do
    current ← pop from S
    if count(A(current)) = 0 then
        V(current) = false
        return false
    end
    foreach neighbor n of current do
        edge ← (current, n)
        pCount ← count(P(edge))
        aCount ← count(A(n))
        P(edge) ← validPlugs(edge)
        A(n) ← validArrangements(n)
        if P(edge) is empty ∨ A(n) is empty then
            V(n) = false
            return false
        end
        else if pCount - count(P(edge)) > 1 ∨ aCount - count(A(n)) > 1 then
            //if anything was removed from P(edge) or A(n)
            push n on S
        end
    end
end
return true
```

Algorithm 1: Constraint propagation

In this algorithm, as discussed in Section 3.3, $P(edge)$ refers to the domain of plug types for a given edge. $V(node)$ refers to whether a node has a verified module.

We make use of two functions, `validPlugs` and `validArrangements` to narrow the possibility space for plug types on edges, and the space for arrangements on nodes.

Determining the valid plugs is simple. For both nodes connected by a given edge, we loop over all arrangements in the node, and look for assignments that refer to the other node. We then insert the plug type belonging to the assignment in a set and return the intersection of the sets for both nodes. Formally we describe that with Equation 3.7.

$$\begin{aligned} validPlugs((i,j)) = \{p(a) | a \in A(i) \wedge n(a) = j\} \\ \cap \{p(a) | a \in A(j) \wedge n(a) = i\} \end{aligned} \tag{3.7}$$

To determine the valid arrangements for node i we need a more complicated approach. We start by assuming all arrangements are valid. We then check in each arrangement A whether:

1. There is an assignment with inconsistent symmetry parameters. Assignments may not specify that a plug has no gender for asymmetric plug types. Formally, $\exists a \in A : \neg isSymmetric(p(a)) \wedge p'(a) = 0$.
2. There an assignment with a plug type not part of the domain of plug types of the edge between the node and the referenced neighbor. Formally: $\exists a \in A : p(a) \notin P((i, n(a)))$
3. There is an asymmetric assignment, for which we can derive that the gender is not consistent with any arrangement of the neighbor. Formally: $\exists a \in A : \neg isSymmetric(p(a)) \wedge (\forall A' \in A(n(a)) : \forall a' \in A' : n(a') \neq a \vee p'(a) = p'(a'))$

If one of these things is the case, we remove the arrangement as a possibility. This is because an arrangement can only be consistent if all assignments are consistent. Technically, the first item is obsolete with our initialization, however, it is a relatively cheap check we kept in for debugging reasons.

3.4.5.1 Termination

As the previously mentioned algorithm shows, we continuously add new neighbors to the stack of nodes to consider next. An important question, in this case, is whether we could say it always terminates. In fact, we can, as long we work in a finite problem space.

Theorem 3.1. *Algorithm 1 will always terminate in finite time if the size of the graph, P and A are finite.*

Proof. A narrowing pass is one run of popping a node from the stack and narrowing all its neighbors. Each narrowing pass needs to either push at least one node on the stack or terminate the algorithm. If a node is pushed on the stack, we know that at least one node or edge has had one value removed from the domain. In a finite problem space, the size of the domain of both arrangements and plug types for each node and edge are also finite. We can, therefore, induce that if the program does not terminate due to a lack of changes, at least one node or edge will eventually reach zero possibilities.

We can simplify this problem by encoding the number of possibilities by node and edge in a vector of integer values. Each narrowing pass, we subtract at least one from at least one element. If we take the sum of all elements in this vector, every pass, this sum should also go down by at least one. If we forbid termination, eventually, this sum would reach zero, which indicates that at least one node or edge has to have zero possibilities, which is a termination condition. We can, therefore, conclude that the algorithm will always terminate. □

Chapter 4

Evaluation

To evaluate the system, we want to give answers to several questions:

1. What is the theoretical performance of our algorithm?
2. What is the practical performance of the generation process, and how does this scale when the problem becomes more complex?
3. Which graph structures are detrimental for the generation performance, and to which extent?
4. Does generation performance depend on the module sets used?
5. Are there clear bottlenecks in the algorithm, and what are those?

In the next section, we give an answer to the first question. In Section 4.2 we set up experiments to give answers to the other questions. However given that the parameter space of the problem is very large, and these parameters are not all independent, we can only give an initial indication.

4.1 Theoretical Performance

Constraint Satisfaction Problems are typically NP-complete problems. Our system is very similar to the AC3 algorithm mentioned in Section 2.3.1. We can show this similarity by mapping the existing problem to a problem in AC3, where the main difference would be the requirement to test arrangements.

The overall system is based on backtracking, Mackworth and Freuder [31] have shown the theoretical performance of such a system, which is $O(\hat{e}\hat{a}^n)$. Here $\hat{e}$ is the number of constraints, $\hat{a}$ the size of the domain (assuming the domain is, before considering domain constraints, the same for all variables) and n the number of variables. Node consistency, thus the constraints that don't involve relations between variables can be ensured in $O(\hat{a}n)$. The constraint propagation of relational constraints, thus AC3, runs in $O(\hat{e}\hat{a}^3)$, with a lower bound of $\Omega(\hat{e}\hat{a}^2)$. This lower bound is when no domain is reduced in the process.

In our system, the variables are the nodes of the graph and the domain is the set of arrangements. The constraints are based on the attributes of these arrangements. The plug type constraints we store on the edge are the first set to consider. Additionally, we consider asymmetry constraints. Our system notably diverges from AC3 by updating the plug type constraints separately, reducing the set of constraints to check against over time, at the cost of having to update this set. Updating this has the same asymptotic complexity as skipping this process, thus it doesn't impact the overall complexity.

Of course, this AC3 algorithm ignores the work we need to do to make a decision of which node we select. The work spent on this scales linearly with the number of nodes and the number of edges per node. As module sets support a limited number of adjacent edges, we can assume a sparse graph, where the number of edges per node on the average is a relatively small constant, we can consider this process linear with the number of nodes.

Testing collision between two module instances depends on the number of box and sphere colliders specified in those modules. For analysis, however, we assume that this number is bound by a small constant. We do not expect very complex compound colliders to be specified with modules. Testing if a given module instance collides with any other instance is, under this assumption, linear with the number of nodes, as we at most need to check against $n - 1$ other instances. Checking if an arrangement fits in the graph is linear with the number of neighbors.

Mackworth and Freuder mentioned that keeping the size of the domain as small as possible is important for making the system fast. Since this is the number of possible arrangements, it is worth figuring out how this number scales in different circumstances.

The number of arrangements at a node is first of all determined by the number of modules. A second aspect is the number of edges starting from a given node. Since we generate arrangements based on the permutations of connector-neighbor assignment, the number of arrangements are thus the factorial of the number of edges multiplied by the number of possible modules. This is however on a single node. To determine the total size of the domain we need to know the size of the union of the domains for each node. Since for each specific number of edges per node, the domains are disjunct (arrangements belonging to nodes with two edges are always different from those that belong to three edges), we can determine the size of the domain for each supported number of edges (the set of which we call J). This would be the number of possible arrangements for the node that supports a given number of edges with the maximum number of modules. This is expressed in Equation 4.1.

$$\hat{a} = |\bigcup_{i \in G} A(i)| = \sum_{j \in J} max_{i \in \{x | x \in G \wedge \#edges(x) = j\}}(|M(i)|) \cdot j! \tag{4.1}$$

For this equation to work, however, arrangements need to be independent of the location in the graph. This is not the case in our representation of an arrangement, as the neighbouring nodes are part of how it is identified. For the purpose of analysis, we can fix that by addressing neighbors by index in a list of neighbors, rather than the neighbors directly. This makes the arrangement specification independent of the location in the graph, allowing it to be reused. This in turn allows for a global domain of arrangements that work on each node.

The overall system is thus as complex as backtracking (without propagation), which was $O(\hat{e}\hat{a}^n)$. However, in practice, we expect that propagation results in reasonable speedups. We can, however, not exclude the possibility that there are cases where propagation has no benefit.

4.2 Experiment Setup

In this section, we look into the practical performance of the algorithm and try to find the bottlenecks in the algorithm. We also look at the diversity of the content produced by the algorithm.

To answer the last four questions mentioned at the beginning of this chapter, we created a set of experiments. In these experiments, we keep track of several metrics that indicate the performance of the algorithm and can give insight in which parts of the algorithm can be a bottleneck. These metrics are the number of iterations, the overall runtime, time spent propagating, checking collisions and in the instantiation phase and finally, we count the reason a module is rejected (fitting failure or collision failure). We executed these experiments on a computer with an AMD Ryzen 1500X CPU and 8GB DDR4 RAM.

We wish to know how performance is affected from several angles. These consist of growing the number of nodes in a graph, but also the structure of the graph and the set of modules that can be used for generation. The hypothesis is that all of these things have an effect on performance.

4.2.1 Module Sets

We use three different module sets, two of which are slight variations of each other, and one being drastically different. The first two work in 2D, and consist of simple rectangles. There are modules for a single end piece, a straight path, a corner, a 3-way split, a 4-way split, and a 180-degree turn. In the first set, these pieces are all colored blue (this is arbitrary). The second set also contains alternative modules for the first five of those, which are colored red. These different modules use a different plug type on their connectors. In addition, we created modules that allow transitioning between the two colors, and thus use both plug types. The third set is modeled to look closer to an action-adventure dungeon and works in 3D. This set has modules for rooms with 1 to 6 exits, with variations in the elevation of an exit. It also has hallway modules, which may contain stairs. They also contain 45 degree turns, which can display that this system doesn't rely on a grid. Finally, there is an empty module with 2 "exits". All module sets are displayed in more detail in Appendix A.

4.2.2 Graphs

In order to test the performance as the size of a graph grows, we test several graphs that follow a simple pattern: A line and a binary tree. For these two structures, we measure performance for 5, 7, 9, 11, 13 and 15 nodes. We also look at two loop-based structures, one being a full loop, and one is a pattern that contains small loops. This pattern alternates a single node connected to a loop of 3 nodes. For these structures, we test 5 and 9 nodes, so we can fairly compare them with the line and binary tree. These graphs are tested on each module set and have no other constraints specified but their structure. The graphs are also visualized in Figure 4.1.

4.2.3 Sample size

We performed a small preparing experiment to evaluate what a feasible sample size is for this experiment. We found that an iteration takes on average 2ms. Based on this we set upper bounds for a single run and determined a sample size of 100 to be the most appropriate. Of course, taking more samples would result in more accurate data, ideally, over 1000 as advised by Arcuri et al. [32], however, 100 should be enough to draw some conclusions. Additionally, in some cases, especially with loop structures, it can take a lot of iterations to find a result, sometimes many more than 100,000 iterations. For this reason, we set a cutoff point at 25,000 iterations. This

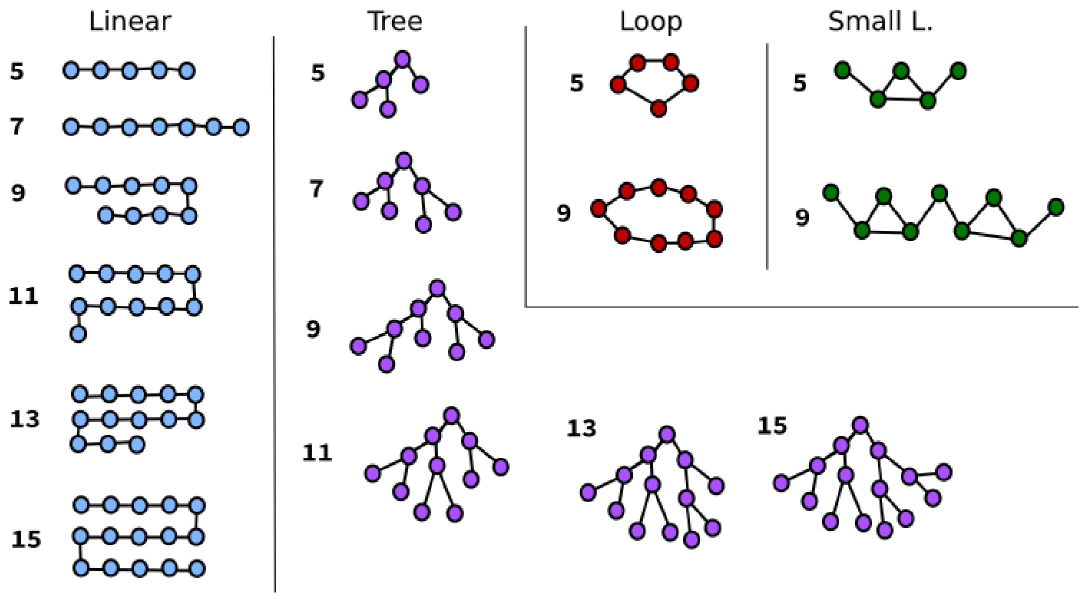


FIGURE 4.1: The graphs we use for our experiments, including how we refer to them in the results section.

would put a cap on the time per run of about 50 seconds with the given execution speed.

4.2.4 Issues

When performing the experiments, we ran into a few issues. The experiments were controlled by an experiment runner script, that would perform the 100 runs in a loop. In some cases (module set 3 on loop structures), the experiment could be busy for more than 5 minutes, after which the runtime massively increases. This is an issue as it meant that subsequent runs might not be independent in performance. We looked into the reason for these issues and we figured out that some experiments can create a massive load on memory. We, therefore, decided to force a garbage collection after each experiment, and give control back to the Unity engine for a bit, which resolved this issue. We however still noticed a high memory load on these structures. We therefore also performed a test on one of these structures (Module 3, Small Loops with 9 nodes) where we force a garbage collection every 500 iterations.

Further, when developing module sets, it appeared to be quite hard to specify connectors perfectly. For this reason, we set a fitting threshold that is quite high, namely 0.1. Modules are one to several units wide.

4.2.5 Statistical Analysis

Of all our experiments we will report standard statistical measures like the mean, median, and standard deviation. We also perform several statistical tests to see if structures and module sets make a difference in performance. Here we test if there is a difference in runtime between module sets for graphs with 5 and 9 nodes. We also test for a difference in structure for these graphs. As data for these comparisons, we split the complete dataset just on the variable to compare. We also leave out any tests not performed on 5 or 9 nodes. As we can not assume that our data is normally distributed, we use the Mann-Whitney U test. As we have multiple tests, 9 in particular, we use a Bonferroni correction ([33], Equation 4.2) on a target overall alpha value of 0.05, resulting in an alpha value of 0.00556 per individual test.

$$\alpha_{individual} = \alpha_{overall}/\text{number of tests} \tag{4.2}$$

Chapter 5

Results

In this chapter, we show the results of the experiments described in the previous chapter. First, we report the success rate and the number of iterations. We then also reflect on the overall runtime of our system, where we also look at the relation between the runtime and number of iterations. We then look at the other metrics to see if they give us some additional insights. Finally, we perform some statistical tests on the data we gathered to determine if module sets and structures make a significant difference.

5.1 Success Rate

First of all, we consider the success rate of the different experiments. This is important as with a low success rate, we can't tell if the other metrics are accurate. In particular, if the success rate is not 100%, the mean of the experiment is skewed downwards, as the number of iterations is limited by an upper bound, and thus the system terminates early at an arbitrarily set point. Because of this, we also report on the median and standard deviation of the different metrics.

As can be seen in Table 5.1, we have a 100% success rate for most of the test runs in module set 1 and 2. Module set 3 is less successful and specifically has an issue with larger graph sizes.

TABLE 5.1: Success Rates

Structure	Set 1	Set 2	Set 3	Structure	Set 1	Set 2	Set 3
Linear 5	100	100	100	Tree 5	100	100	100
Linear 7	100	100	100	Tree 7	100	100	100
Linear 9	100	100	100	Tree 9	100	100	100
Linear 11	100	100	100	Tree 11	100	100	100
Linear 13	100	100	90	Tree 13	100	100	100
Linear 15	100	100	79	Tree 15	100	100	100
Loop 5	100	100	70	Small L. 5	100	100	99
Loop 9	96	77	7	Small L. 9	100	100	0

Number of successful generation attempts out of 100 runs, by graph structure and number of nodes. The cutoff point is 25000 iterations.

5.2 Number of Iterations

As mentioned before, we mentioned the number of iterations for each module set and various graphs. In Figure 5.1 we display the mean number of iterations for each set of experiments.

We can see that the loop structures are notably slower than the structures without loops. We can also see a super-linear growth in the number of iterations as graphs become larger in those loop-less structures. In Table B.1 we show the exact metrics (mean, median and standard deviation).

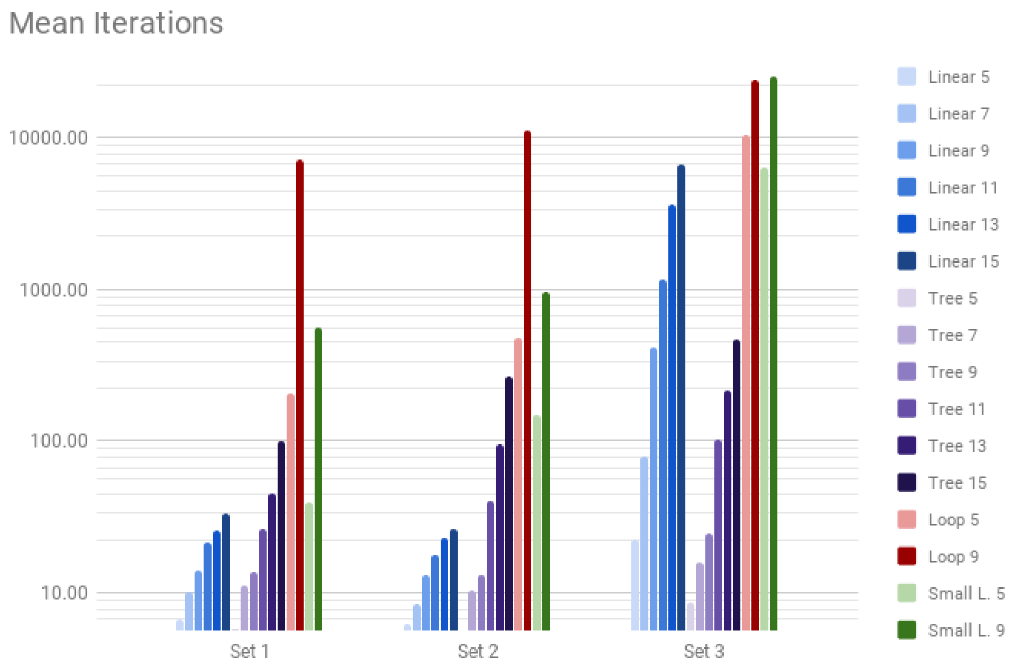


FIGURE 5.1: Chart of the mean iterations, on a logarithmic scale.

5.3 Runtime

We also measured the actual time it took to run these experiments. The mean performance is displayed in Figure 5.2, which is also displayed, along with the median and standard deviation in Table B.2. The runtime figures follow the same trend as the number of iterations.

The performance seems to scale super-linearly. We observed a large difference between the graphs with loops in them and the graphs without loops. The full loop structure can cause a mean runtime performance impact ranging from a factor 15 to a factor 2222 when comparing to linear and tree graphs with the same number of nodes. The small loops structure has less of an impact, with a range of factor 3 to 596.

5.3.1 Relation Between Runtime and Iterations.

We also compared the relation between runtime and iterations. This is displayed in Table B.3 and Figure 5.3. Here we can see that the speed at which iterations are executed are variable. Module set 3 is notably slower, but also more consistent. For module set 2 and 3, the speed is, on average, actually faster for the larger tree structures and the loop structures compared to the linear structures. Finding the reason for this requires a more thorough investigation.

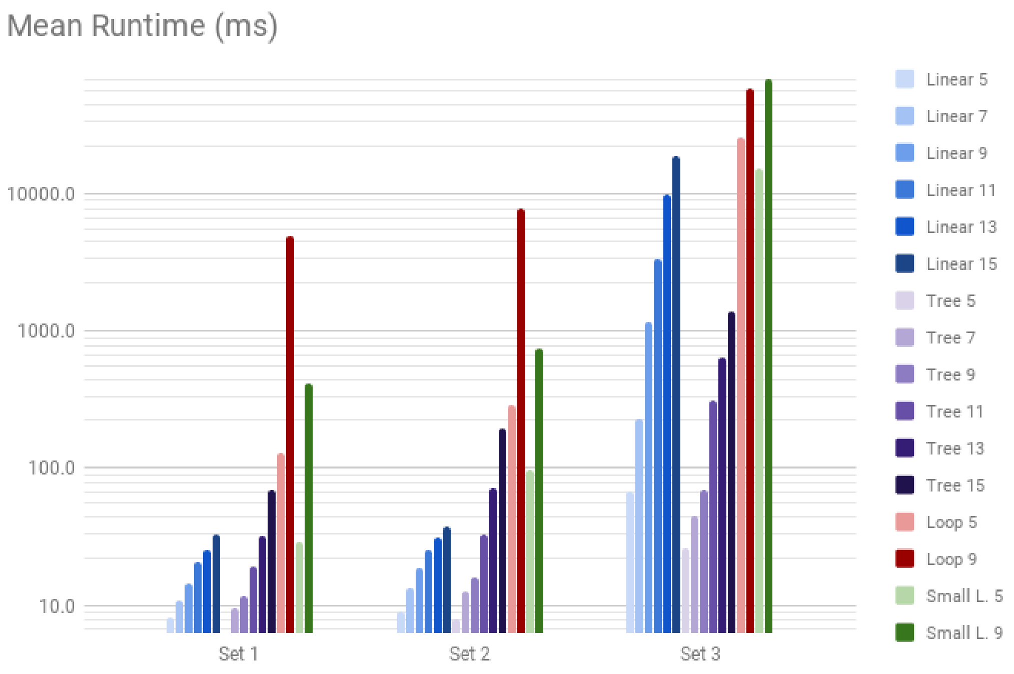


FIGURE 5.2: Chart of the mean runtime, on a logarithmic scale.

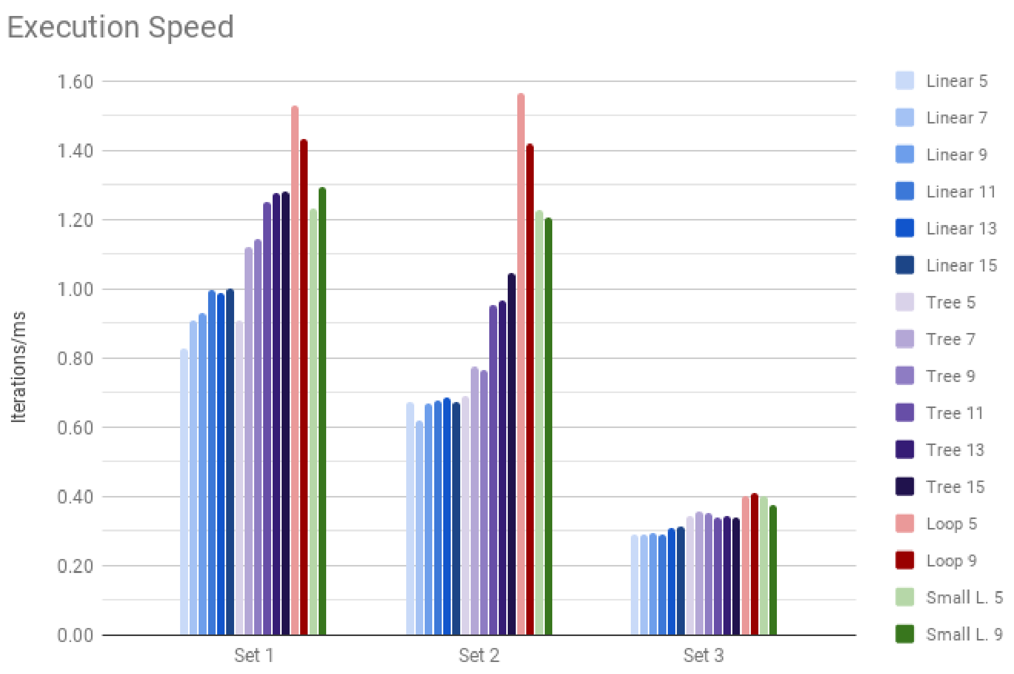


FIGURE 5.3: Chart that displays the mean execution speed of the algorithm in iterations per millisecond.

5.3.2 Statistical Analysis

We performed statistical analysis to determine if there is a significant difference in runtime (medians) between module sets and structures. For this we performed the Mann-Whitney U-test, using the implementation in R. In particular we used the function `wilcox.test(x,y, paired= FALSE, conf.inf= TRUE)`. Calling the function with `conf.inf = TRUE` also provides us with an effect measure: The Hodges-Lehmann estimation. This is an estimation of how much the medians are expected to differ. We limited the tests to graphs with 5 and 9 nodes. For module set comparison this means that each structure is equally represented, avoiding a bias towards the loop-less structures, and for structure comparison, it is required as we lack other data for the loop structures. As mentioned in Section 4.2.5, we use an α-value of 0.00556 per test, determined by the Bonferroni correction.

Since the Mann-Whitney U test is based on quantiles, we also report the quantiles for each of the compared data sets.

5.3.2.1 Module Set Comparison

TABLE 5.2: Quantiles Module Set Runtime Comparison

	Minimum	1st Quartile	Median	3rd Quartile	Maximum
Set1	5.5	8.7	15.7	134.1	18391.9
Set2	7.2	12.9	24.1	287.9	18740.5
Set3	10.4	33.0	568.4	59913.6	116558.0

Quantiles of the runtime data containing all experiments on graphs with 5 and 9 nodes, split by module set.

TABLE 5.3: Module Set Runtime Comparison

	W	p-value	Hodges–Lehmann estimation
1-2	262760	5.83E-10	-4.435637
1-3	123490	<2.2e-16	-329.2924
2-3	153760	<2.2e-16	-295.5611

Comparison between module sets based on data from experiments on graphs with 5 and 9 nodes. We used the Mann-Whitney U-test with a null hypothesis of equality.

From the data we see in Table 5.3 we can conclude that for the compared data sets, there is a significant median difference between module sets.

5.3.2.2 Structure Comparison

TABLE 5.4: Quantiles Structure Runtime Comparison

	Minimum	1st Quartile	Median	3rd Quartile	Maximum
Linear	6.2	8.9	15.7	27.7	17808.2
Tree	5.5	7.7	13.1	20.5	478.4
Loop	7.3	180.6	1977.3	17626.8	69066.4
Small L.	7.2	50.8	389.9	12770.3	116558.0

Quantiles of the runtime data containing all experiments on 5 and 9 nodes, split by graph structure

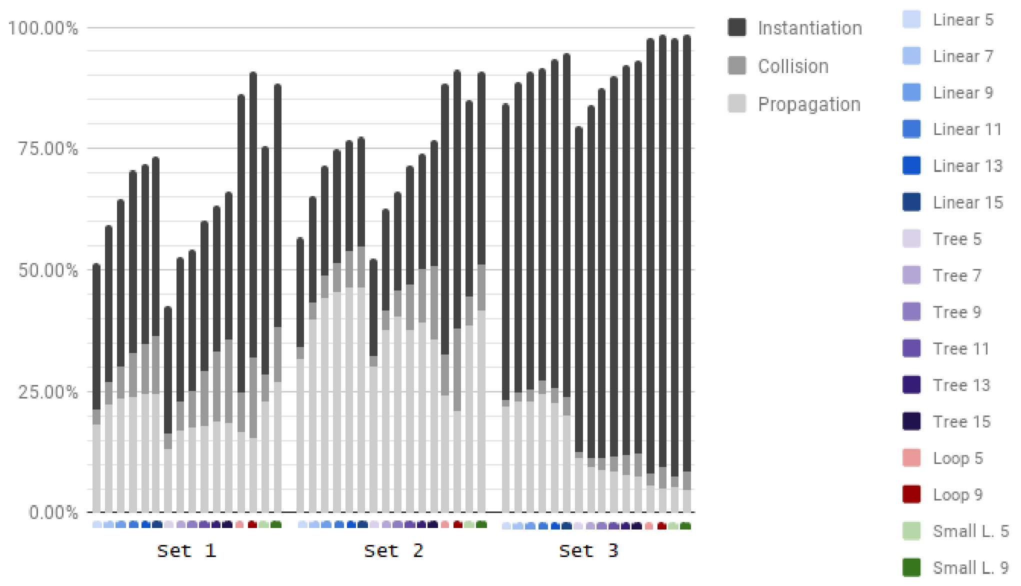


FIGURE 5.4: Chart that displays the mean time distribution between the different parts of the algorithm.

TABLE 5.5: Structure Runtime Comparison

	p-value	Hodges–Lehmann estimation
Linear vs Tree	1.51E-07	2.168266
Linear vs Loop	<2.2e-16	-1752.036
Linear vs Small L.	<2.2e-16	-334.8532
Tree vs Loop	<2.2e-16	-1956.731
Tree vs Small L.	<2.2e-16	-366.8003
Loop vs Small L.	2.39E-08	179.3259

Comparison between structures based on data from experiments on graphs of 5 and 9 nodes. We used the Mann-Whitney U-test with a null hypothesis of equality.

The data that were shown in Table 5.5 also indicates a significant median difference between structures on graphs with 5 and 9 nodes, as all the p-values are below our alpha value of 0.00556.

5.4 Profiling

In addition to measuring runtime, we also measured the runtime of some key components of the algorithm. These are the constraint propagation system, the instantiation system, and the collision detection system. Not explicitly measured are the remaining parts of the algorithm which consist of the decision model and the backtracking system which includes creating backup states. In Figure 5.4 we display the average time distributions between the different parts of the algorithm. Exact numbers including median and standard deviation can be found in Tables B.6, B.5 and B.4. Interesting is that in case of module set 3, a very large part of the time is spent on instantiation when compared to the other two module sets. For module sets 1 and 2, especially for smaller graph sizes, a lot of time is spent on the not explicitly measured part of the algorithm.

5.5 Validation Decisions

During the validation process, an arrangement can be either accepted or rejected for two reasons: a fitting failure or a collision failure. In Appendix B.3 we present tables that show the frequency of these failures. In particular, fitting failures should only occur for loop structures, however, the mean and standard deviation is not 0 for module set 3 on the linear structure. Looking at the actual data, we found that there were a number of runs that had a single fitting failure. The reason for this could be a bug involving an edge case for fit checking or module placement, however, even in those cases, we would expect multiple fitting failures to occur. We, therefore, cannot properly explain the reason for this anomaly.

Another thing that is worth noting is that Loop 5 has no collision failures whatsoever on Module sets 1 and 2. And having very few collision failures is actually not that rare, as there are 11 other tests that have a median of 0 collision failures.

5.6 The Effect of Garbage Collection

We performed an extra test with a forced garbage collect every 500 iterations to resolve a memory issue that is most prevalent with the Small Loops structure with 9 nodes. It resulted in an average of 65351.6ms in runtime, and a median of 66175.3. Comparing this to the experiment without garbage collection (67968.9ms and 64403.9ms respectively), we can not conclude that it had any difference. We did, however, observe a lower memory usage over the runtime. Of course, garbage collection also costs some overhead, so a significant runtime improvement would have been surprising. Full results of the extra test is displayed in Appendix B.4.

5.7 Generation Output

Of course, it is also valuable to know what the output of the generation process is. While we do not have a full evaluation on diversity, we can show some examples of the output, which we present Appendix C. Here we also include a few examples for graphs we did not perform experiments with.

Chapter 6

Conclusion and Future Work

In this thesis we considered a constraint solving approach to 3D level generation for structures not restricted to a grid. We made some steps to develop a system that is capable of doing so and evaluated the performance of the current process. Based on this evaluation, we can suggest future work to optimize the system. This evaluation, however, is not complete, more extensive evaluation can be done, especially in terms of output analysis. It also remains a question whether the system could be useful in practice. We further reflect on the design issues we encountered with this system and look into ways that said issues can be resolved.

When reflecting back on our results, we can see that the runtime seems to scale super-linearly with the number of nodes, although more data is needed to get to a conclusive answer. We also noticed that loop structures do have a large performance impact. We also observed that module sets influence the performance, and this impact is quite significant. Since this impact also seems to influence the difference between structures, it is hard to give a conclusive answer to the extent at which the structures and module sets differ, and even harder to generalize this to other structures and module sets. One thing we did notice is that in the case of module set 3, a surprisingly large amount of time is spent in the initialization phase. Also, the unmeasured overhead was larger than we expected, which was more clear on module sets 1 and 2.

6.1 Further Evaluation

While we measured the performance for various graphs and module sets, there are far more possibilities and we only scratched the surface of practical performance testing. In our theoretical analysis, we found that the number of edges per node should have an impact on performance, in addition to the number of modules possible on said nodes. A controlled test where these factors are varied could be used to verify if this holds in practice. Another point of interest is to look at the impact of defining constraints on nodes and edges. The current evaluation focuses only on the topological constraints, or constraints that exist purely due to the graph definition. Performing the tests with a larger number of samples, like 1000 as mentioned in Chapter 4, would also allow for a stronger conclusion.

Profiling itself can also be made more extensive, with a focus on smaller pieces of code.

6.1.1 Diversity Analysis

Due to the scope of the project, we have not been able to provide an analysis of the diversity of the generated content. Creating such an analysis is, however, valuable to assess the usefulness of the system. It can be quite tricky to determine the proper metric. However, we can specify a few requirements for a good metric:

- The metric should be independent of representation. With symmetric graphs, the same dungeon can be represented in different ways.
- The metric should capture differences in arrangements, taking symmetry of modules into account.
- The metric should ensure that subgraphs that result in similar structures need to be recognized as similar.

Creating a metric that meets all these requirements is an issue. One could compare pairs of modules to create a representation independent metric, which also meets the third requirement, but it doesn't capture arrangements. Considering symmetry in arrangements can be even harder, as we don't consider symmetry in our system at all. Another issue is that even when one can compare two results, we still need a method to determine a meaningful metric of the diversity of the complete experiment. In case of the module-pair metric, we can determine the standard deviation of each pair and then sum that, but this is still a metric that depends on the input graph, thus making it hard to compare the diversity between different graphs.

In specific implementations, it could also be possible to classify modules and determine metrics based on the frequency of said classifications. Examples would be the difference between rooms and hallways or 2D and 3D structures in module set 3. It would then be possible to evaluate the output using an Expressive Range plot, as introduced by Smith [34]. This approach is commonly used to evaluate the output of generation processes.

6.2 System Improvements

6.2.1 Optimization Suggestions

From our experiments, we found that for certain module sets, the instantiation part of the algorithm can become a significant bottleneck. It would be worth restricting this instantiation part down to the essentials (skeleton and collision), and only instantiating the geometry after the fact. In this implementation, the collision model would no longer be (optionally) tied to the geometry. We can ensure this at module specification, however it could also be done as a pre-processing step. Further, as the same modules are likely used over and over again, even on the same node, it could be worth it to keep an object pool of modules to test with rather than instantiate new modules every iteration and destroy modules each rollback. This is valuable as the destroyed modules need to be cleaned up, which does not seem to happen automatically given the memory issues we had.

Further, on the first two module sets the part that is not explicitly measured is also a significant part of the computation time. Additional investigation would be required to see what the bottleneck is in that phase.

6.2.2 Partial Generation

While we investigated the application of generating content from scratch, the nature of the algorithm should be well suited for partial generation problems. If one has a previously generated problem, it could be possible to remove some modules, modify parts of the graph, and solve this part of the problem again. Still, existing modules could be initialized as verified and made immutable (cannot be destroyed in the process), and the modified system should be able to fill the gaps.

This case should create far more varied generation frontiers, possibly allowing it to take more advantage of the constraint propagation and decision selection process. It would be interesting to evaluate this application.

6.2.3 System Design

We noticed that the problem as it is can become quite complex due to the difference in modules. It might be worth considering splitting certain parts of the algorithm into separate steps. In particular, it could be interesting to select modules based on shape first, and then solve based on plug types once the spatial structure is known, similar to how Spelunky first generates a global path, before filling it in with templates. The resulting problem can then take advantage of more advanced constraint solving problems, such as path consistency and WFC (if we can properly provide patterns), and we no longer have to test whether modules fit in this stage. The two-phase approach also allows for multiple levels to be generated off the same spatial structure, thus skipping a part of the process.

6.3 Summary

In this thesis, we described a new 3D level generation system based on constraint solving. This is a system that allows designers to have a large amount of control over the output of the system, and express this control in a declarative way by constraining the topology of the level with a graph. In the graph, additional constraints can be specified to add additional control. The system searches for rigid modules and orientations to connect them together in a way that is consistent with the graph. This system may fit in a larger generation pipeline where also graphs can be generated.

We explored whether this approach can be used to deal with a weakness of grammar-based approaches with long looping structures. However, from our results, we have indications that these long structures perform quite poorly as well. Additional work is required to optimize the algorithm to deal with this weakness. We identified several parts of the algorithm that are worth focusing optimization efforts on and made a few suggestions. We also identified that both module sets and graph structures seem to have an impact on performance, although we cannot explain why. For that, we need more extensive evaluation.

Finally, we considered that the algorithm, with a few modifications, could also be applied for partial generation. This can be useful for assisting a designer to create levels by filling in gaps and evaluating constraints.

Appendix A

Module Sets

In this Appendix, we show our module sets. In Table A.1 we display the number of modules that support a given number of connectors for each set. In Figures A.1 through A.3 we show the geometry of the modules and their names. The colliders on the modules are taken generously, especially for set 3. Many colliders don't extend outside the geometry, this allows walls to clip a bit. The reason for this is that we didn't want to encounter collision failures due to slight misalignments.

TABLE A.1: Number of modules by the number of connectors

	Set 1	Set 2	Set 3
1-way modules	2	3	4
2-way modules	4	8	17
3-way modules	1	3	2
4-way modules	1	3	1
5-way modules	0	0	1
6-way modules	0	0	1
Total modules	8	17	26

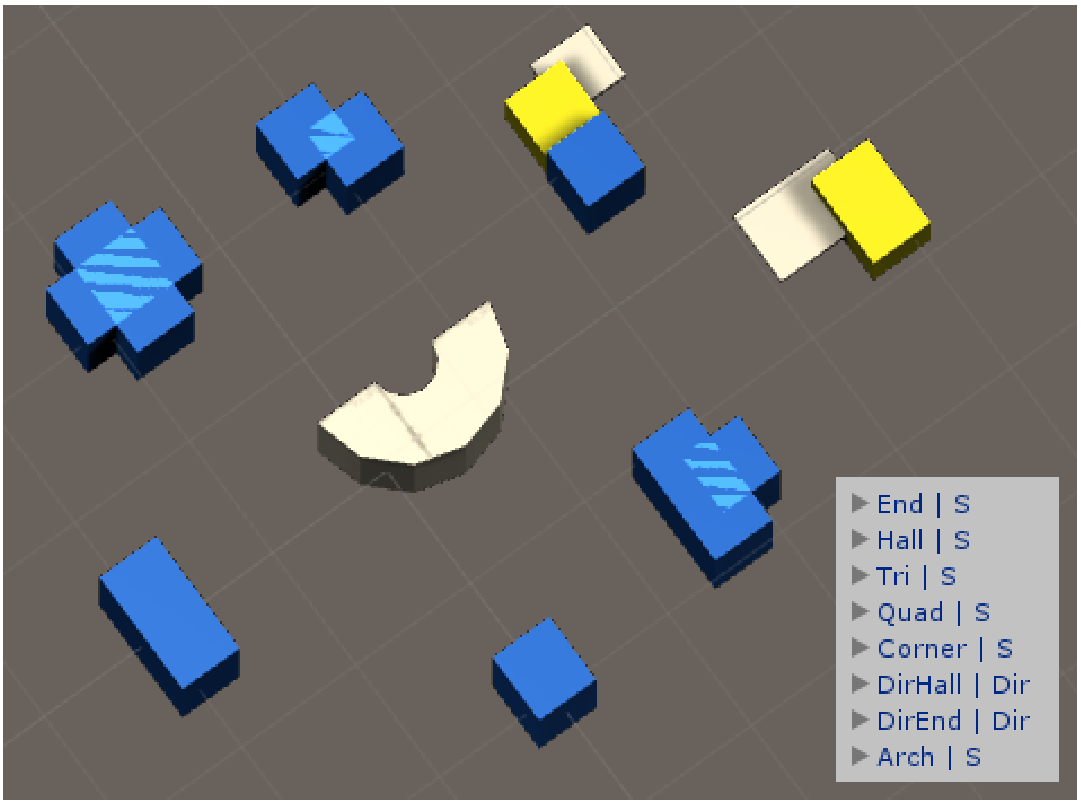


FIGURE A.1: Module set 1 geometry. All plug types are equal except for the yellow objects, which use a different and asymmetric plug type. The white arc object uses the same plug type as the blue objects.

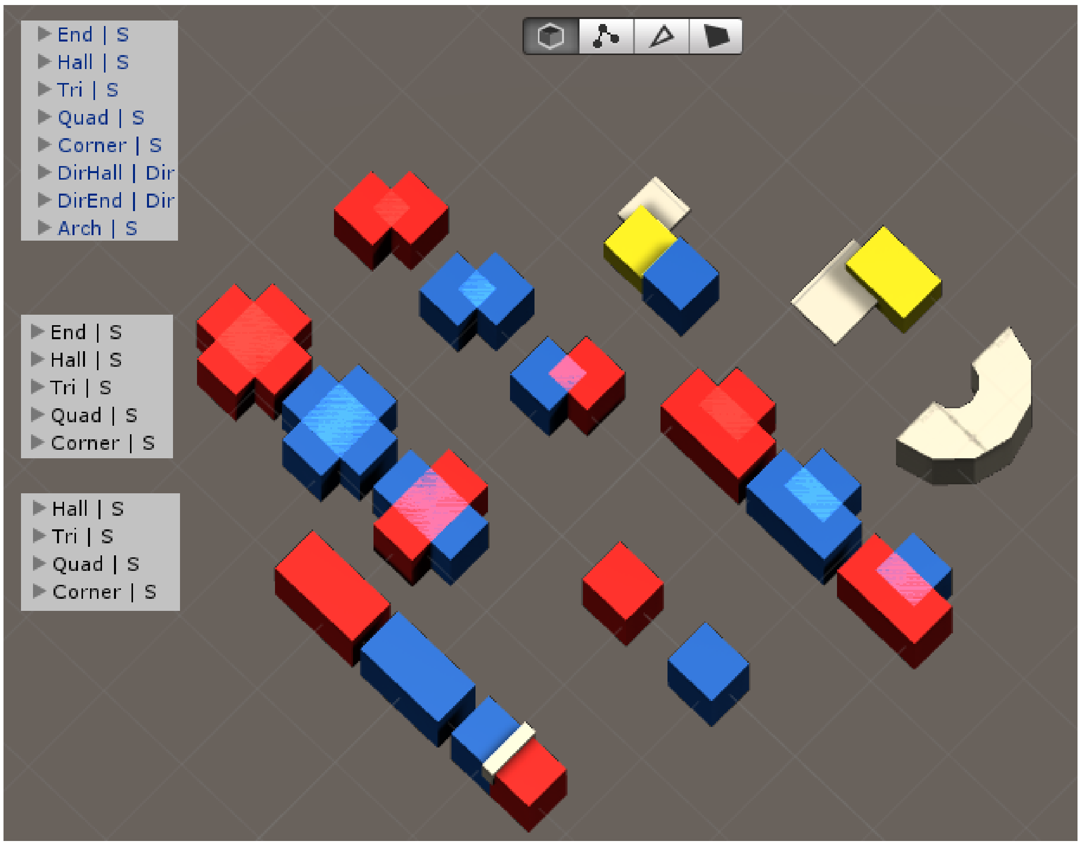


FIGURE A.2: Module set 2 is an extension of the first set. This set introduces the red pieces, which connect with a different symmetric plug type. We also introduce pieces that connect these different types.

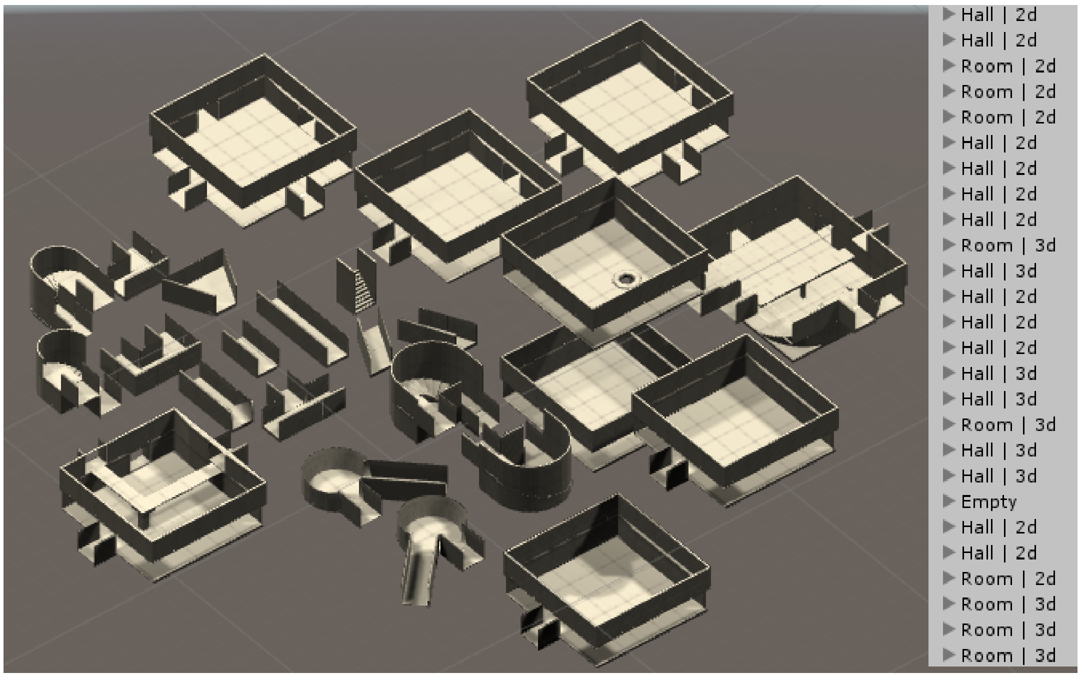


FIGURE A.3: The third set is made to resemble typical room-based dungeons more. It has one default plug type to connect with, and one asymmetric one to drop down.

Appendix B

Result Tables

In this appendix, we show more detailed results from the experiments.

B.1 Iterations

TABLE B.1: Iteration Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	6.56	6.21	22.48	2.75	2.83	43.26	6	5	7
Linear 7	10.15	8.34	78.97	4.28	2.83	202.19	8	7	10
Linear 9	13.80	12.93	408.35	5.25	5.71	1113.76	11.5	10	16
Linear 11	21.37	17.66	1162.27	8.71	8.83	3650.37	19	13.5	17
Linear 13	25.89	22.64	3586.02	13.75	12.67	7862.87	22.5	21	33
Linear 15	32.88	26.36	6702.29	12.78	15.74	10246.76	30	24	76
Tree 5	5.73	5.59	8.57	1.14	1.85	6.43	5	5	6
Tree 7	10.95	10.27	15.64	4.38	5.25	12.41	8	8	9
Tree 9	13.75	12.91	24.49	8.07	8.23	24.31	10	10	13.5
Tree 11	26.05	39.88	102.03	23.24	52.69	242.24	18	16.5	26
Tree 13	45.06	94.56	215.30	65.21	189.02	366.62	21	21.5	47.5
Tree 15	99.52	267.14	462.96	162.05	517.14	818.48	31.5	40	62
Loop 5	202.34	476.91	10349.74	178.80	514.34	10477.00	154.5	305	4190.5
Loop 9	7126.12	11150.42	23787.88	7992.48	9676.44	4845.20	3506.5	8658.5	25000
Small L. 5	39.27	146.82	6282.27	35.67	171.49	5274.80	22	85	5256.5
Small L. 9	555.78	956.26	25000.00	535.97	791.96	0.00	386	831.5	25000

B.2 Runtime

TABLE B.2: Runtime Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	8.1	9.0	67.8	5.0	2.0	116.9	7.0	8.2	27.0
Linear 7	10.8	13.3	225.6	2.9	2.9	552.1	9.5	12.2	40.9
Linear 9	14.5	18.7	1155.1	3.6	4.6	3122.9	13.3	16.4	61.6
Linear 11	20.9	25.1	3324.4	6.0	7.4	10448.0	19.3	22.7	68.2
Linear 13	25.6	31.4	9972.1	10.4	11.2	21862.0	23.1	29.1	116.8
Linear 15	32.5	37.3	19007.5	11.2	12.9	29006.7	29.1	34.6	234.9
Tree 5	6.3	8.0	26.0	0.8	1.1	22.5	6.1	7.6	19.0
Tree 7	9.5	12.7	45.2	2.5	3.4	36.5	8.4	11.4	25.4
Tree 9	11.6	15.9	70.3	4.9	5.1	70.2	9.4	14.1	38.4
Tree 11	19.0	33.0	310.6	12.8	30.3	741.5	14.4	19.7	77.7
Tree 13	32.0	71.0	634.2	39.1	117.4	1086.1	17.2	26.2	137.7
Tree 15	69.6	195.1	1385.9	105.5	345.2	2454.3	24.5	44.2	201.3
Loop 5	127.8	287.6	25597.2	111.3	296.5	26066.2	97.7	187.7	10094.3
Loop 9	4885.1	7730.5	57731.0	5479.4	6727.6	11875.5	2429.1	6070.3	60371.9
Small L. 5	28.6	97.7	15474.4	22.2	98.3	13032.1	17.8	67.0	12900.9
Small L. 9	408.7	741.7	67968.9	381.7	591.2	11340.7	293.3	660.0	64403.9

B.2.1 Execution Speed

We also looked at the speed at which iterations are executed in iterations per millisecond.

TABLE B.3: Execution Speed Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	0.83	0.67	0.29	0.15	0.11	0.05	0.80	0.64	0.29
Linear 7	0.91	0.62	0.29	0.13	0.10	0.05	0.87	0.60	0.28
Linear 9	0.93	0.67	0.30	0.13	0.12	0.04	0.91	0.62	0.29
Linear 11	1.00	0.68	0.29	0.13	0.12	0.05	1.01	0.62	0.28
Linear 13	0.99	0.69	0.31	0.12	0.12	0.05	0.99	0.69	0.30
Linear 15	1.00	0.67	0.31	0.08	0.12	0.04	1.00	0.67	0.33
Tree 5	0.91	0.69	0.34	0.08	0.08	0.05	0.89	0.68	0.34
Tree 7	1.12	0.77	0.36	0.14	0.17	0.05	1.06	0.69	0.35
Tree 9	1.14	0.77	0.35	0.14	0.15	0.04	1.08	0.70	0.35
Tree 11	1.25	0.95	0.34	0.21	0.32	0.03	1.21	0.84	0.34
Tree 13	1.27	0.97	0.34	0.17	0.31	0.03	1.22	0.84	0.34
Tree 15	1.28	1.04	0.34	0.16	0.30	0.03	1.25	0.93	0.33
Loop 5	1.53	1.57	0.40	0.16	0.20	0.03	1.58	1.62	0.41
Loop 9	1.43	1.42	0.41	0.10	0.11	0.02	1.45	1.44	0.41
Small L. 5	1.23	1.23	0.40	0.24	0.40	0.02	1.24	1.30	0.41
Small L. 9	1.29	1.20	0.38	0.11	0.19	0.05	1.33	1.26	0.39

B.2.2 Instantiation Times

Here we report on the percentage of the runtime that is spent in the instantiation phase of the algorithm. This is the part after a decision has been made, until it is verified or rejected, excluding the time spent checking collisions.

TABLE B.4: Instantiation Time Percentage Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	30.04%	22.63%	61.19%	7.05%	6.45%	12.90%	29.06%	19.38%	58.45%
Linear 7	32.33%	21.78%	63.96%	7.22%	5.08%	13.64%	31.16%	20.83%	60.84%
Linear 9	34.47%	22.40%	65.76%	5.96%	5.82%	13.19%	35.01%	20.84%	63.75%
Linear 11	37.63%	23.41%	64.37%	5.36%	6.25%	13.27%	37.62%	23.81%	60.94%
Linear 13	37.03%	22.84%	67.81%	5.33%	5.66%	13.86%	37.21%	22.77%	67.64%
Linear 15	37.06%	22.65%	70.73%	5.08%	5.56%	12.95%	37.61%	22.83%	75.30%
Tree 5	26.46%	19.92%	66.97%	3.77%	2.90%	8.90%	24.84%	19.19%	66.66%
Tree 7	29.71%	20.96%	72.74%	4.90%	4.69%	9.27%	27.21%	18.36%	70.33%
Tree 9	29.01%	20.47%	76.27%	3.88%	4.22%	6.71%	26.53%	18.31%	75.35%
Tree 11	30.91%	24.51%	78.70%	4.68%	7.91%	6.04%	30.59%	21.97%	78.67%
Tree 13	30.19%	23.92%	80.36%	3.84%	7.92%	6.06%	29.33%	21.08%	81.04%
Tree 15	30.28%	25.94%	81.05%	3.82%	7.18%	5.51%	30.08%	23.55%	81.45%
Loop 5	61.44%	55.94%	89.71%	6.97%	6.95%	6.52%	62.00%	57.95%	91.62%
Loop 9	58.86%	53.37%	88.87%	4.08%	3.93%	3.07%	59.95%	54.45%	89.29%
Small L. 5	47.28%	40.38%	90.19%	9.53%	9.91%	3.53%	49.12%	42.37%	91.02%
Small L. 9	50.28%	39.76%	90.19%	5.87%	5.85%	1.17%	51.80%	41.32%	89.78%

B.2.3 Collision Checking Times

Here we report on the percentage of the time spent checking for collisions.

TABLE B.5: Collision Checking Time Percentage Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	3.13%	2.50%	1.36%	1.47%	0.87%	0.98%	2.74%	2.21%	1.10%
Linear 7	4.81%	3.44%	1.86%	1.26%	0.91%	0.74%	4.24%	3.18%	1.60%
Linear 9	6.46%	4.81%	2.36%	1.58%	1.45%	0.85%	6.14%	4.23%	2.07%
Linear 11	9.07%	6.20%	2.73%	2.78%	1.72%	0.80%	8.83%	5.55%	2.50%
Linear 13	10.24%	7.62%	3.30%	2.32%	2.38%	1.03%	10.22%	6.79%	3.05%
Linear 15	12.00%	8.37%	3.88%	2.42%	2.58%	1.17%	11.57%	7.55%	3.52%
Tree 5	3.04%	2.30%	1.33%	0.56%	0.48%	0.67%	2.95%	2.25%	1.22%
Tree 7	5.91%	4.17%	2.12%	1.69%	1.69%	1.15%	5.03%	3.31%	1.94%
Tree 9	7.56%	5.37%	2.54%	1.92%	2.10%	0.59%	6.60%	4.49%	2.45%
Tree 11	11.21%	9.26%	3.12%	3.45%	4.61%	0.67%	9.97%	7.30%	2.98%
Tree 13	14.34%	10.90%	3.80%	4.42%	5.24%	0.75%	13.34%	8.49%	3.68%
Tree 15	17.23%	14.97%	4.56%	5.09%	6.74%	0.93%	15.31%	12.55%	4.54%
Loop 5	8.36%	8.56%	2.30%	2.05%	1.28%	0.33%	8.18%	8.81%	2.29%
Loop 9	16.74%	16.88%	4.48%	1.98%	1.69%	0.56%	16.94%	17.23%	4.59%
Small L. 5	5.46%	5.83%	2.45%	1.89%	2.05%	0.28%	5.47%	6.14%	2.48%
Small L. 9	11.35%	9.41%	3.69%	3.57%	2.89%	0.44%	11.70%	9.61%	3.83%

B.2.4 Propagation Times

In Table B.6 we report the percentage of the runtime spent propagating constraints.

TABLE B.6: Constraint Propagation Time Percentage Statistics

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	18.14%	31.75%	21.79%	1.63%	3.09%	6.76%	18.12%	32.17%	23.01%
Linear 7	22.14%	39.94%	22.81%	1.56%	2.83%	8.71%	22.40%	40.36%	24.18%
Linear 9	23.60%	44.24%	22.88%	1.79%	4.04%	9.00%	23.45%	44.01%	24.29%
Linear 11	23.73%	45.35%	24.54%	1.78%	3.86%	9.66%	23.71%	45.18%	26.90%
Linear 13	24.44%	46.27%	22.45%	2.83%	4.06%	10.58%	24.32%	47.12%	23.55%
Linear 15	24.33%	46.50%	19.95%	2.78%	4.15%	9.71%	24.34%	46.51%	13.96%
Tree 5	13.20%	30.03%	11.24%	1.50%	2.91%	3.16%	13.61%	29.26%	11.31%
Tree 7	16.91%	37.67%	9.27%	2.02%	2.74%	2.50%	16.93%	38.01%	9.27%
Tree 9	17.55%	40.31%	8.68%	2.97%	3.18%	1.98%	16.96%	40.57%	8.68%
Tree 11	17.96%	37.64%	8.31%	2.67%	6.37%	2.27%	17.58%	40.41%	8.10%
Tree 13	18.75%	39.29%	7.98%	2.80%	7.19%	2.30%	17.83%	42.34%	7.85%
Tree 15	18.58%	35.87%	7.55%	3.23%	10.17%	2.45%	17.41%	37.15%	7.38%
Loop 5	16.54%	23.97%	5.72%	2.80%	3.22%	4.57%	16.00%	23.04%	4.42%
Loop 9	15.25%	21.00%	4.94%	2.19%	3.10%	2.33%	14.61%	19.98%	4.55%
Small L. 5	22.92%	38.62%	5.19%	3.43%	7.66%	1.64%	22.46%	36.47%	4.86%
Small L. 9	26.87%	41.65%	4.61%	6.54%	6.98%	0.72%	24.91%	39.66%	4.75%

B.3 Validation Results

In this section we report metrics about the number of times validation failures occurred in the algorithm. We distinguish two kinds of failures: collision failures and fitting failures. In the case of collision failures, an arrangement is rejected due to overlap. In the case of a fitting failure, an arrangement is rejected because connectors don't align as they should.

B.3.1 Collision Failures

TABLE B.7: Collision-based validation failures

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	1.02	0.91	15.70	2.31	2.46	40.47	0	0	1
Linear 7	2.44	1.02	66.84	3.61	2.45	190.06	0	0	2
Linear 9	3.85	3.17	374.96	4.51	4.99	1047.87	2	0	6
Linear 11	8.56	5.58	1082.94	7.39	7.79	3437.06	7	1.5	4.5
Linear 13	10.81	8.21	3362.66	11.70	11.05	7402.87	7.5	6.5	20
Linear 15	15.15	9.71	6292.99	10.86	13.68	9644.44	13	8	54.5
Tree 5	0.17	0.12	2.48	0.47	0.66	5.70	0	0	0
Tree 7	2.25	1.74	6.61	3.13	3.66	10.18	0	0	1
Tree 9	2.82	2.69	12.19	5.49	6.35	19.38	0	0	3
Tree 11	8.26	18.66	74.90	13.74	36.27	199.44	3	2	12.5
Tree 13	18.29	53.07	162.03	40.80	121.04	294.62	6	3.5	29.5
Tree 15	43.81	134.93	354.26	82.16	255.23	645.04	7	13	40.5
Loop 5	35.09	34.44	1424.26	46.83	53.61	1716.38	17.5	14	695.5
Loop 9	1775.85	1523.06	9257.10	1995.62	2093.36	7144.55	814.5	824	6471
Small L. 5	0.00	0.00	1936.87	0.00	0.00	1815.82	0	0	1328.5
Small L. 9	317.26	366.87	5988.77	400.05	461.56	4603.36	56.5	72	4111

B.3.2 Fitting Failures

TABLE B.8: Fitting-based validation failures

	Mean			Standard Deviation			Median		
Structure	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Linear 5	0.00	0.00	0.14	0.00	0.00	0.35	0	0	0
Linear 7	0.00	0.00	0.15	0.00	0.00	0.36	0	0	0
Linear 9	0.00	0.00	0.15	0.00	0.00	0.36	0	0	0
Linear 11	0.00	0.00	0.18	0.00	0.00	0.39	0	0	0
Linear 13	0.00	0.00	0.18	0.00	0.00	0.39	0	0	0
Linear 15	0.00	0.00	0.14	0.00	0.00	0.35	0	0	0
Tree 5	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Tree 7	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Tree 9	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Tree 11	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Tree 13	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Tree 15	0.00	0.00	0.00	0.00	0.00	0.00	0	0	0
Loop 5	159.63	362.42	9976.46	144.38	392.80	10108.75	121	231	4055
Loop 9	5482.31	8294.26	22409.50	6299.73	7093.75	4587.18	2718.5	6683.5	23561.5
Small L. 5	28.38	105.12	6020.69	29.28	125.49	5066.04	15	63.5	5036.5
Small L. 9	366.00	624.72	23947.41	348.83	514.02	314.81	252	578.5	24000

B.4 Miscellaneous

Finally, we also show the results of the test we did where we force a garbage collect every 500 iterations. These results are discussed in Section 5.6.

It consists of 100 runs on the Small Loops structure with 9 nodes.

TABLE B.9: Experiments with Garbage Collection

	Mean	Std. Dev	Median
Iterations	24451.2	3222.9	25000.0
Runtime	65351.6	8841.6	66175.3
Propagation Time	5.34%	0.75%	5.14%
Collision Time	3.97%	0.30%	4.05%
Instantiation Time	88.46%	0.58%	88.56%
Collision Failures	6375.3	4909.6	4515.5
Fitting Failures	23275.5	3109.8	23974.5

Metrics for experiments with Garbage Collection every 500 iterations. In this set of experiments we had 3 successful runs.

Appendix C

Output Examples

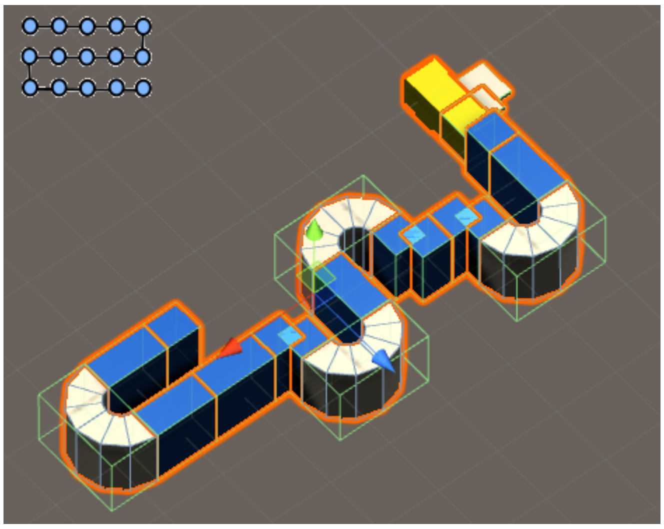


FIGURE C.1: A generation output with module set 1 for a linear graph with 15 nodes.

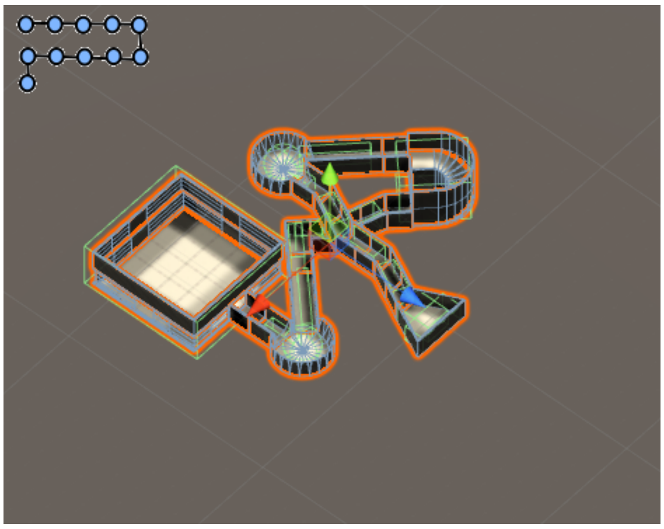


FIGURE C.2: A generation output with module set 3 for a linear graph with 11 nodes.

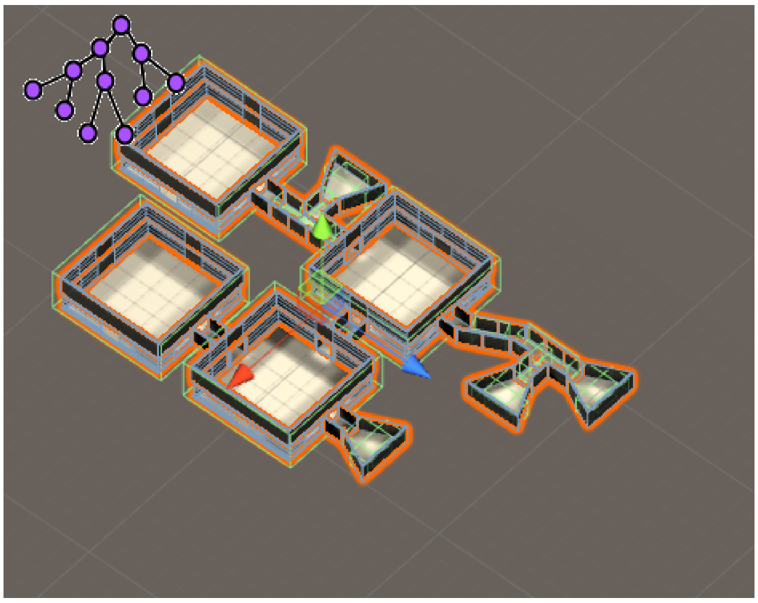


FIGURE C.3: A generation output with module set 3 for a binary tree graph with 11 nodes.

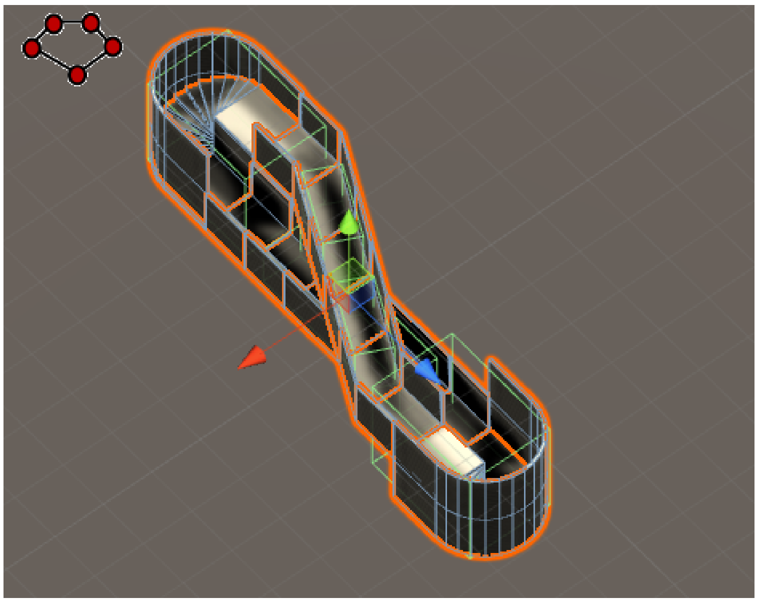


FIGURE C.4: A generation output of module set 3 for a loop graph with 5 nodes.

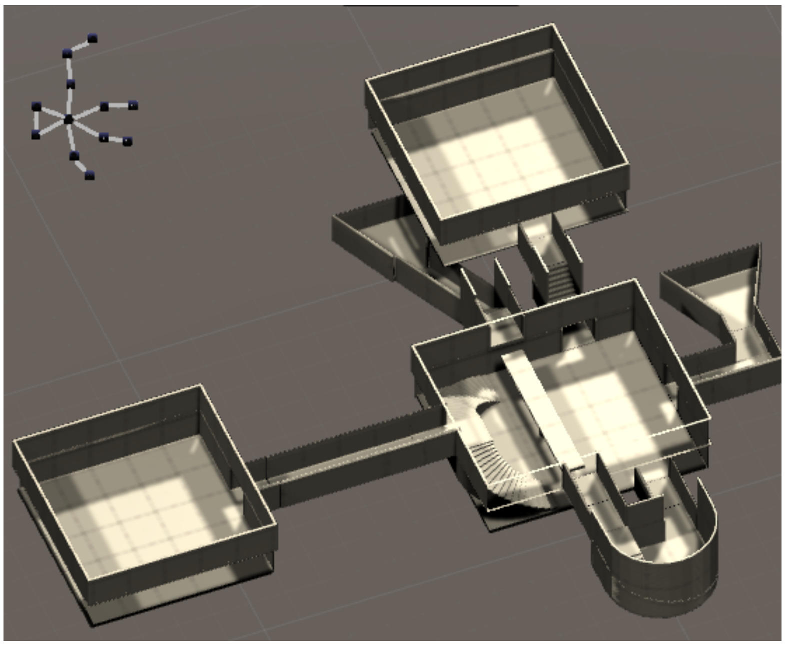


FIGURE C.5: A generation output of module set 3 for a graph that contains a node with 6 adjacent edges.

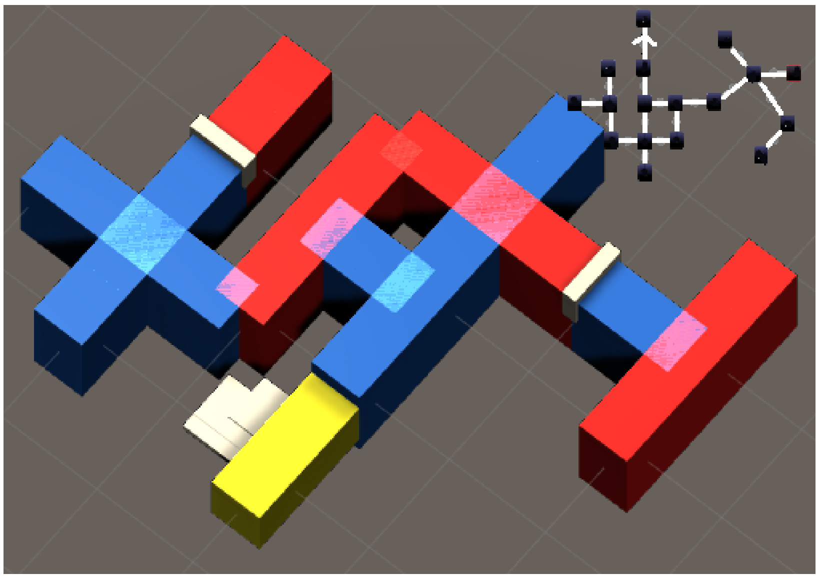


FIGURE C.6: A generation output of module set 2 on a graph with 17 nodes that contains both a loop and branches.

Bibliography

[1] Alan K Mackworth. "Consistency in Networks of Relations". In: *Artificial Intelligence* 8.1 (1977), pp. 99–118.

[2] Alba Amato. "Procedural Content Generation in the Game Industry". In: *Game Dynamics*. Springer, 2017, pp. 15–25.

[3] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[4] Derek Yu. *Spelunky*. Mossmouth, LLC, 2008.

[5] Cyrille Bonard. *Ruggnar*. Early Access. Swords N' Wands, 2018.

[6] Joris Dormans. "Adventures in level design: generating missions and spaces for action adventure games". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM. 2010, p. 1.

[7] Oskar Stålberg, Richard Meredith, and Martin Kvale. *Bad North*. Plausible Concept AB, 2018.

[8] Isaac Karth and Adam M Smith. "WaveFunctionCollapse is constraint solving in the wild". In: *Proceedings of the 12th International Conference on the Foundations of Digital Games*. ACM. 2017, p. 68.

[9] Maxim Gumin. *WaveFunctionCollapse (source)*. https://github.com/mxgmn/WaveFunctionCollapse. 2018.

[10] GameScience. *Refraction*. Kongregate, 2010.

[11] Nintendo. *The Legend of Zelda (series)*. Nintendo, 1986-2017.

[12] Nintendo. *The Legend of Zelda: Skyward Sword*. Nintendo, 2011.

[13] Mr-DeKay. *Zelda - Skyward Sword: Skyview Temple Map*. https://www.deviantart.com/mr-dekay/art/Zelda-Skyward-Sword-Skyview-Temple-Map-290051069. Accessed 28-11-2018.

[14] A Adonaac. *Procedural Dungeon Generation Algorithm*. http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php?print=1. 2015.

[15] Darius Kazemi. *Spelunky Generator Lessons*. http://tinysubversions.com/spelunkyGen/. Accessed 28-10-2018. 2013.

[16] Cyrille Bonard. *Level Generation in Ruggnar*. https://ruggnar.com/ProcGen/. Accessed 28-10-2018. 2018.

[17] Noam Chomsky. "Three models for the description of language". In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124.

[18] Przemyslaw Prusinkiewicz. "Graphical applications of L-systems". In: *Proceedings of Graphics Interface*. Vol. 86. 86. 1986, pp. 247–253.

[19] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[20] David Adams. *Automatic generation of dungeons for computer games*. 2002.

[21] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. "Designing procedurally generated levels". In: *Proceedings of the the second workshop on Artificial Intelligence in the Game Design Process*. 2013.

[22] Bart Middag, Peter Lambert, and Sofie Van Hoecke. *Controllable Generative Grammars for Multifaceted Generation of Game Levels.* 2016.

[23] Leif Foged and Ian Horswill. "Rolling Your Own Finite-Domain Constraint Solver". In: *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. AK Peters/CRC Press, 2015.

[24] University of Potsdam. *Clingo*. `https://github.com/potassco/clingo`. 2018.

[25] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. "A case study of expressively constrainable level design automation tools for a puzzle game". In: *Proceedings of the International Conference on the Foundations of Digital Games*. ACM. 2012, pp. 156–163.

[26] Paul Merrell and Dinesh Manocha. "Model synthesis: A general procedural modeling algorithm". In: *IEEE transactions on visualization and computer graphics* 17.6 (2011), pp. 715–728.

[27] Oskar Stålberg. *Wave (WFC mixed-initiative showcase)*. `http://oskarstalberg.com/game/wave/wave.html`.

[28] Oskar Stålberg. "Wave Function Collapse in Bad North". `https://www.youtube.com/watch?v=0bcZb-SsnrA`. Everything Procedural at Breda University of Applied Sciences. 2018.

[29] Joseph Parker and Ryan Jones. *Proc Skater*. Itch.io, 2016.

[30] Freehold Games. *Caves of Qud*. 2016.

[31] Alan K Mackworth and Eugene C Freuder. "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems". In: *Artificial Intelligence* 25.1 (1985), pp. 65–74.

[32] Andrea Arcuri and Lionel Briand. "A practical guide for using statistical tests to assess randomized algorithms in software engineering". In: *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE. 2011, pp. 1–10.

[33] Eric W. Weisstein. *Bonferroni Correction*. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/BonferroniCorrection.html`. Accessed 25-10-2018.

[34] 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*. ACM. 2010, p. 4.