Spatial Layout of Procedural Dungeons Using Linear Constraints and SMT Solvers

Jim Whitehead
ejw@ucsc.edu
Univ. of California, Santa Cruz
Santa Cruz, CA

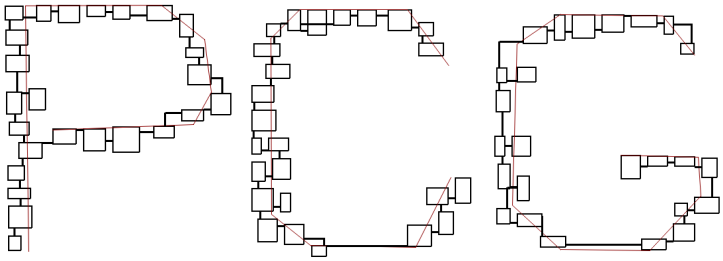

Figure 1: Using PCG to create dungeons.

ABSTRACT

Dungeon generation is among the oldest problems in procedural content generation. Creating the spatial aspects of a dungeon requires three steps: random generation of rooms and sizes, placement of these rooms inside a fixed area, and connecting rooms with passageways. This paper uses a series of integer linear constraints, solved by a satisfiability modulo theories (SMT) solver, to perform the placement step. Separation constraints ensure dungeon rooms do not intersect and maintain a minimum fixed separation. Designers can specify control lines, and dungeon rooms will be placed within a fixed distance of these control lines. Generation times vary with number of rooms and constraints, but are often very fast. Spatial distribution of solutions tend to have hot spots, but is surprisingly uniform given the underlying complexity of the solver. The approach demonstrates the effectiveness of a declarative approach to dungeon layout generation, where designers can express desired intent, and the SMT solver satisfies this if possible.

CCS CONCEPTS

• **Applied computing** → **Computer games**; • **Software and its engineering** → **Interactive games**; • **Computing methodologies** → **Spatial and physical reasoning**.

KEYWORDS

procedural content generation, dungeon generation, rectangle packing, linear constraints, satisfiability modulo theories, SMT

FDG '20, September 15–18, 2020, Bugibba, Malta

ACM ISBN 978-1-4503-8807-8/20/09.
https://doi.org/10.1145/3402942.3409603

ACM Reference Format:
Jim Whitehead. 2020. Spatial Layout of Procedural Dungeons Using Linear Constraints and SMT Solvers. In *International Conference on the Foundations of Digital Games (FDG '20), September 15–18, 2020, Bugibba, Malta.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3402942.3409603

1 INTRODUCTION

Dungeons are a common element in fantasy themed computer games. In a dungeon crawler, the player proceeds through a series of dungeon rooms, battling monsters, collecting treasure, and solving puzzles. *Rogue*, one of the early games to feature procedural content generation, is a dungeon crawler with generated dungeons. It demonstrated the feasibility of automated dungeon generation, and inspired an entire genre of Roguelike games where adventuring through a procedural dungeon is a defining characteristic [17]. Generally viewed as underground labyrinths, dungeons come in two main forms: a built environment consisting of a set of rooms connected by passageways, or a natural environment composed of a series of interconnected caverns. The focus of this paper is on built environment dungeons.

Procedural dungeon generation consists of up to three broad phases: (1) *mission:* determining the dungeon's goals and puzzles (especially lock and key puzzles)[13][32], (2) *room layout:* generating rooms and placing them in space consistent with the mission, and (3) *interior decoration and layout:* final placement of monsters, furnishings, collectibles, and treasure. Many dungeon generators lack the mission phase, due to its complexity. Interior decoration and layout can be viewed as an example of the *interior layout* (or *scene synthesis*) problem [35][21].

We view the room layout phase as consisting of three steps: selection, layout, and connection. *Room selection* begins with room selection. Rooms are either selected from a library of pre-existing rooms, or have randomly generated sizes. In either case, the result is a collection of dungeon rooms of varying sizes. *Layout* involves placing these rooms within a fixed-size rectangular playfield

area which limits the maximum size of the dungeon. The key challenge in dungeon layout is placing rooms in the playfield while avoiding overlaps. Once rooms have been placed, the *connection* step connects them with passageways (if separate) or doorways (if touching)[8].

The need to place rooms without overlap has driven existing approaches for dungeon layout. In *space partitioning* approaches, the playfield is subdivided into non-overlapping cells, and then a room is placed entirely inside each cell[30]. In the original Rogue, space was partitioned into a 3x3 grid and a room was randomly generated in each grid cell, with some cells left empty. Other approaches involve use of space partitioning trees of varying kinds (e.g., binary space partition trees). This approach has the drawback of forcing rooms to be located wholly inside cells to ensure lack of overlap. Placing a room such that it overlaps a cell boundary is not allowed, which limits the possible locations where a room can be placed. Complex arrangements of rooms is typically not possible, such as multiple rooms with varying sizes in close proximity to each other. This is because the space partitioning algorithm typically cannot guarantee that a specific set of grid shapes will be located next to each other.

In *random walk* approaches, an agent randomly traverses the playfield, placing rooms or passageways during its travels. While allowing for greater flexibility in room locations, it has the drawback of placement bias, since rooms will more frequently be placed close to the start location [7][16][30]. Other placement approaches include *force directed* layout, where rooms all begin in the center of the playfield, and they "push" against one another in an iterative physics simulation until there are no overlaps [1]. This tends to locate rooms in a rough circle around the center of the playfield, avoiding corners. The *place and test* approach randomly places a room, checks for intersections, and then re-places as necessary [8]. The drawback is the exponential increase in re-place steps as room coverage increases, and the possibility of creating (and inability to detect) unsolvable layouts. This is addressed in [8] by limiting the total number of placement tries, and accepting the resulting number of rooms.

An over-arching problem with existing dungeon layout approaches is *lack of designer control*. Two simple scenarios highlight this. Consider a designer who wishes to place a large throne room in the middle of the playfield, taking up 20% or more of the entire space. An antechamber must be located by the front entrance, and an escape room must be located near the thrones. Guaranteeing satisfaction of these constraints is beyond the capabilities of existing layout approaches. Space partition grids would need additional mechanism to ensure proximity constraints are met, and might have trouble creating the appropriate set of partitions. Random walk and force-directed approaches lack awareness of constraints such as these. The second design scenario involves a designer providing a series of "control lines" which describe a path through the playfield with dungeon rooms to be placed such that they are in proximity to the control lines. Existing approaches are incapable of guaranteeing rooms will be placed along such control lines.

To ensure high degrees of designer control, the dungeon placement approach must meet several requirements. It must be possible to specify basic constraints, such as freedom from overlap. More complex designer constraints must be expressible. Rooms should have complete freedom of placement location, subject to any constraints upon them. The layout algorithm must be capable of simultaneously satisfying basic and designer constraints, or indicate these constraints are unsatisfiable. Finally, the layout algorithm should be fast, and have relatively uniform placement of rooms in allowable locations.

We view dungeon layout as a constraint satisfaction problem, where the goal is to generate sets of valid solutions to a system of constraints on allowable room locations. Constraints are generally linear in form, that is, constraints take the form of linear equations generally involving the x,y location of the upper left corner of a room. This paper explores use of the Z3 SMT solver to perform dungeon layout using linear constraints. SMT stands for *satisfiability modulo theories*, and an SMT solver can be viewed as a boolean satisfiability solver (a SAT solver, the "satisfiability" part of the name) which can additionally handle other classes of constraints. These other constraint types each have different algorithms for reasoning about and solving them—the general name "theory" is used to describe these algorithms. Hence, "satisfiability modulo theories" can be viewed as boolean satisfiability plus additional theories. The "satisfiability modulo theories" name is unfortunate, since it's needlessly opaque and hides its true nature as a general purpose constraint solver. Z3 is a specific constraint solver which supports many different constraint types[1]. It has been under development by Microsoft Research for over 8 years (open source since 2015), has received multiple awards, and is in widespread use across many projects [12].

This paper makes the following contributions. It expresses the dungeon layout problem as a constraint satisfaction problem, and describes the necessary basic and designer linear constraints. An implementation of this approach is described using the Z3 SMT solver. It demonstrates how an SMT solver can address a long-standing procedural content generation problem, and provide significantly enhanced designer control over the outcome. The behavior of this implementation is explored, focusing on execution speed and the frequency distribution of solutions.

2 RELATED WORK

2.1 Dungeon Room Layout

Several authors provide descriptions of "classical" techniques for dungeon room layout. Van der Linden et al. provide the first survey of dungeon and cavern generation techniques, including cellular automata, generative grammars, and genetic algorithms [39]. Shaker et al. provide an introduction to constructive techniques for dungeon generation, including space partitions and random walks, as well as use of cellular automata for cavern generation [30]. Baron separates room placement (random and binary space partition) and corridor creation (random point connect, random walk, binary space partition), and empirically explores combinations of these [7]. Williams describes binary space partitions, delaunay graphs, and cellular automata to generate dungeons [25].

Dormans introduced the idea of first generating the goals and puzzles of a level in the form of a "mission graph" and thereby introducing the mission phase of generation. This mission graph

[1] https://github.com/Z3Prover/z3/wiki

is used to control physical dungeon layout [13]. Van der Linden explored this idea further by generating action graphs, and then using these to drive physical dungeon layout for the game Dwarf Quest [29]. Karvalos et al. build on this idea, and demonstrate mission and dungeon generation using a pipeline of generative grammars using the Ludoscope tool [10]. Lavender and Thompson also explore the idea of dungeon generation using mission graphs for an open-source Zelda-like game [23].

Some researchers have explored evolutionary algorithms for dungeon creation. In the Sentient Sketchbook by Liapis et al., a genetic algorithm generates different level alternatives which are presented to the use in a mixed-initiative design tool. One interpretation of these levels is as dungeons [3]. Baldwin et al. created a mixed-initiative dungeon generator for Zelda-like levels where dungeon concepts are initially generated from a catalog of design patterns [5]. An evolutionary algorithm creates four new dungeon ideas which are presented to the designer for possible adoption. The paper has a rich set of metrics for evaluating levels. Valtchanov and Brown create a graph structure representing a dungeon layout, and then evolve these graphs to find increasingly large and complex dungeons. Graphs are converted into rooms via a placement algorithm that prunes sub-trees when collisions are detected [38]. Liapis et al. use evolutionary search to generate dungeons for the Roguelike game Minidungeons. Different player personas (monster killer, treasure collector, speedrunner, survivalist) were used to develop evolved player controllers. These, in turn, were used to guide evolutionary development of dungeon levels, leading to different level geometry for each persona type [24].

A growing body of research focuses on declarative dungeon generation and layout. Adam Smith used Answer Set Programming (ASP) to declaratively describe features of cavern-type dungeons, including playability constraints such as reachability of the exit [31]. Roden and Parberry describe a level design approach involving constraints expressed over a graph structure, but without detail on the expression of the constraints or the constraint solving approach [28]. Anthony Smith and Joanna Bryson use ASP for the room selection, layout, connection and decoration steps of dungeon generation, but do not use mission graphs [2]. In contrast, recent work by T. Smith et al. generates a mission graph using ASP, but does not focus on physical layout [32]. The focus of this ASP work on using declarative programming to generate caverns and dungeons is aligned with our focus on declaratively specifying layout constraints using SMT.

Though a detailed comparison of the ASP and SMT approaches is beyond the scope of this paper, we can mention a few tradeoffs. For boolean and linear constraints, both SMT and ASP appear to have similar expressive power, though the Z3 solver appears to have a greater range of constraints it can handle beyond these. Syntactically, ASP requires learning a new programming language (AnsProlog), while Z3 supports multiple programming language bindings which allow constraints to be expressed via API calls. The author finds the AnsProlog language to be awkward and frustrating to use, but notes others are more enthusiastic; personal taste might decide whether the AnsProlog or API approach is preferred. The Z3 language bindings also make it easier to transfer data to/from the constraint solver using Z3. The Z3 solver generally performs random sampling of the solution space, as compared to ASP which outputs answer sets in order, requiring a further sampling step to pull random instances. A detailed comparison of ASP vs SMT for equivalent dungeon generation tasks would be interesting, including performance trade-offs.

2.2 Similar Layout Problems

Placing furniture within a room, subject to constraints, is known as the *interior layout problem*. Interior layout is very similar to dungeon room layout: placement of furniture shapes (dungeon rooms) inside the walls of a room (playfield) without overlaps, subject to distance and adjacency (and control line) constraints. Interior layout was an early application domain for constraint solving, an example being Pfefferkorn's 1975 CACM paper [26]. This paper describes furniture layout subject to minimum distance, room quadrant, orientation, adjacency, no overlap, viewable, and path reachable constraints, which are resolved by a custom constraint solver implemented in Lisp. Later researchers brought furniture layout under the more general umbrella of *declarative modeling* where designers provide declarative descriptions (in the form of constraints) of how they want a 3D scene to be composed from 3D objects, with solutions determined by a constraint solver. Gaildrat [40] and Tutenel et al. [36] provide an overview of issues involved in declarative modeling, and specific examples of declarative modeling systems include [9][22][37]. Within the PCG community, Balint and Bidarra present an approach for generating present-day room interiors that could be used for dungeon room decoration [6], and Tutenel et al. provide a constraint-based approach for interior layout [35].

Placing rooms within the walls of a building, subject to constraints, is variously known as the *room layout*, *building layout*, *space layout planning*, or *spatial layout* problem. Architectural room layout is similar to dungeon room layout, in that both involve placement of rooms within a fixed envelope, with two key differences. Architectural rooms are expected to completely fill the shape defined by the building walls while dungeons rooms only partially fill the playfield. Architectural rooms are adjacent to one another without overlaps, while dungeon rooms may either be adjacent or separate from each other, also without overlap. Lobos and Danath (2010) provide architectural requirements for space layout planning, and trace computational approaches back to 1955 [11]. They survey both academic and commercial approaches. Hamouda provides a survey of space layout planning from 1965-2000, with greater focus on constraint-based approaches [18]. The computer games community started exploring this problem in 2006 with Hahn et al.'s paper on view-specific building interiors [14], and Martin's approach which generates a layout graph using graph grammars, then expands the volume of each room via a "pressure" model [20]. Lopes et al. expand on Martin's ideas, describing a multi-stage approach involving hierarchical subdivision of building spaces, resolution of adjacency constraints, and a final room expansion phase [27]. Recent work in architecture has explored the use of neural networks (mostly convolutional neural networks) for architectural room layout [19][33].

Our formulation of the dungeon layout problem is very similar to the classic *bin packing (rectangle packing)* problem. However, bin packing is typically framed as finding an *optimal* packing of items into a space. For dungeon generation, we typically wish

rooms to be spread out across the space to maximize designer and gameplay goals. Stoykov explored use of SMT solvers for the rectangle packing problem in his Masters thesis, and developed linear constraints similar to those in this paper [34].

Overall, dungeon room layout differs from these other layout problems primarily in the type of constraints. As compared to the majority of constraint solving approaches for layout problems, this paper uses an off-the-shelf SMT constraint solver, rather than implementing a constraint solver from scratch. Ideally this reduces the gap between the declarative specification of constraints and their solution.

3 DUNGEON ROOM LAYOUT USING LINEAR CONSTRAINTS

The dungeon room layout problem is formally described as follows. Given a set of N rooms where each room is r_i with varying widths ($width_i$) and heights ($height_i$) and a playfield with dimensions $play_w$ and $play_h$, place each r_i so that it is fully inside the playfield, and no room overlaps one another. Each room has a location (x_{r_i}, y_{r_i}) defined to be the upper left hand corner of the room, assuming a typical game coordinate system with the origin in the upper left, with x-axis increasing to the right and y-axis increasing downwards (typical for computer graphics). In the approach described herein, rooms are limited to be rectangular in shape (instead of arbitrary polygons), cannot be rotated (have fixed orientation), and must be axis-aligned.

Linear constraints are expressions between one or more variables that take the form of one of the following. There are no exponents, logarithms, square roots, etc.

$$\begin{aligned} c_1x_1 + c_2x_2 + \ldots c_nx_n &\leq 0 \\ c_1x_1 + c_2x_2 + \ldots c_nx_n &= 0 \\ c_1x_1 + c_2x_2 + \ldots c_nx_n &\geq 0 \end{aligned}$$

In practice, the Z3 SMT solver permits constraints to be expressed in any form, not just equations with a zero on one side. Division is also permitted in Z3, though this has significant negative impacts on performance (perhaps due to using a different decision procedure within Z3). Coefficients can be negative and hence each of the additions above can be read as "addition or subtraction". Coefficients are limited to integers, which can be challenging for expressing slopes (see discussion below in control lines). These constraint equations can be connected together with logical and & or.

With this notation in place, we can proceed to the basic and designer constraints.

3.1 Basic Constraints

Within-playfield constraints ensure that every room is placed inside the playfield:

$$\begin{aligned} \forall r_i \in Rooms, x_{r_i} >= 0 \wedge x_{r_i} < play_w - width_i \wedge \\ y_{r_i} >= 0 \wedge y_{r_i} < play_h - height_i \end{aligned} \tag{1}$$

The starting location needs to be greater than or equal to zero, and less than the width/height of the playfield minus the width/height of the individual room. Note that each of these conditions is connected with a logical "and", meaning that all of these constraints must hold for the overall within-playfield constraint to be satisfied.

Separation constraints ensure that every room does not overlap another room, and each room maintains a specified separation distance, sep, from all other rooms. Given two rooms, i and j, there are 8 possible positions that j can have relative to i (above left, above center, above right, adjacent left, adjacent right, below left, below center, below right). However, this can be expressed as room j is located either above or below or left or right of room i, where above, below, left, and right are:

$$\begin{aligned} above &: y_{r_j} \leq y_{r_i} - height_j - sep \\ below &: y_{r_i} \leq y_{r_j} - height_i - sep \\ left &: x_{r_j} \leq x_{r_i} - width_j - sep \\ right &: x_{r_i} \leq x_{r_j} - width_i - sep \end{aligned} \tag{2}$$

To specify separation in the form of constraints, it is necessary to use *logical or* to connect all of the individual direction constraints together: $above \vee below \vee left \vee right$ Filling in the exact constraints, this takes the form: $y_{r_j} \leq y_{r_i} - height_j - sep \vee y_{r_i} \leq y_{r_j} - height_i - sep \vee x_{r_j} \leq x_{r_i} - width_j - sep \vee x_{r_i} \leq x_{r_j} - width_i - sep$.

These separation constraints need to be established once for every unique pair of rooms, $i, j, i \neq j$, hence for N rooms this results in $N(N-1)/2$ sets of separation constraints.

3.2 Logical Or Creates a Generative Space

It is worth thinking about the implications of the use of logical or in the placement constraints above. For two rooms, these four placement constraints (connected by logical or) create four unique possibilities for their relative placement. In combination with the playfield constraints, the large number of possible starting positions for room i, combined with the four possible relative placements for j, combine to create a very large number of possible placements of these two rooms. That is, the or'ed together positional constraints act to define different generative possibilities. As the number of rooms increases, the number of possible relative placements increases as well, defining an increasingly large generative space of valid placements. Every time we see a logical or, it defines a binary choice the constraint solver can make, and hence a different possible generative outcome.

3.3 Design Constraints

We revisit the two design scenarios from the introduction to highlight the flexibility of linear constraints for expressing designer constraints.

Throne room. The designer wishes to have a large throne room placed towards the middle of the playfield, with an antechamber immediately below it, and an escape room located near the thrones at the top. Achieving this goal involves (a) setting a placement constraint for the throne room, (b) ensuring the antechamber is the same width as the throne room, (c) constraining the location of the antechamber relative to the throne room, and (d) constraining the location of the escape room relative to the throne room. Constraints (c) and (d) are used in lieu of the basic separation constraints, since

they provide different kinds of separation. The throne room design constraints take the form of:

$$\begin{aligned}
&(a)\ throne_x \geq 0.3play_w \wedge throne_x \leq 0.35play_w \wedge throne_y \\
&\quad \geq 0.1play_h \wedge throne_y \leq 0.25play_h \\
&(b)\ antechamber_{width} = throne_{width} \\
&(c)\ throne_y = antechamber_y - throne_{height} \wedge throne_x \\
&\quad = antechamber_x \\
&(d)\ throne_x = escape_x - throne_{width} \wedge throne_y = escape_y \\
&\quad -0.1throne_{height}
\end{aligned} \tag{3}$$

Examples of dungeons generated using these constraints can be found in Figure 2.

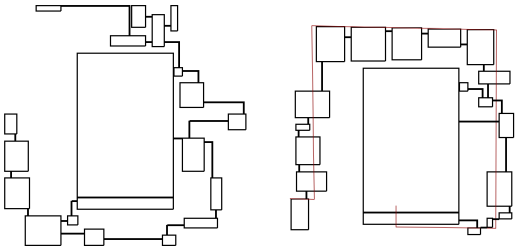

Figure 2: Representative generated throne rooms. The throne room is the large room in the center, the antechamber is the same width an immediately below. A small escape room is connected to the upper right of the throne room. The right figure shows the throne room and control line constraints (red lines) active at the same time.

Control Lines. The designer gives a series of (connected) line segments and wants the dungeon rooms to be placed within a fixed line width of these control lines. Figure 1 shows an example of dungeons generated using control lines. These lines are defined by a start and end point, such as when using a series of mouse clicks to describe these lines, and hence use the two-point equation for a line to define our line constraints.

$$\begin{aligned}
&r_{i_y} \leq slope * (r_{i_x} - end_x + linewidth) + end_y \wedge \\
&r_{i_y} \geq slope * (r_{i_x} - end_x - linewidth + width_i) + end_y
\end{aligned} \tag{4}$$

For negative slopes we switch ≤ and ≥ comparators. Z3 limits us to integer slope values when using integer constraints, which caused floating point slope values to be truncated. This was unfortunate since many useful slopes lie between 0 and 1, and these were truncated to 0. As a workaround, we scaled up just the y dimension by a scaling factor of 1000. In this way, a slope that had been, say, 0.01549 would now be 15 (instead of 0 before scaling), a close enough approximation. Values are de-scaled before being displayed or used in-game. We explored an alternative approach where the numerator and denominator of the slope were used in the equation instead of a pre-computed slope, but this introduced a single divide which had the effect of slowing solution times by several orders of magnitude.

Without further constraints, the lines run across the entire playfield, and do not stop at the start and end points. We need to add a constraint preventing values before the start, and after the end. While we need to constrain both x and y, we only need to constrain one variable since the other is constrained via the linear relationship.

$$r_{i_y} \geq start_y \wedge r_{i_y} \leq end_y - height_i \tag{5}$$

When the line is close to horizontal, the fall of the line over its length is less than the room height, and hence it is impossible to place a room. In this situation, we switch to use x-based constraints instead.

4 EMPIRICAL EVALUATION

To evaluate these ideas, we implemented a dungeon generator in Python using the Z3 constraint solver. Our dungeon generation process begins by randomly generating a series of rooms, selecting random widths and heights within designer-specified bounds. These rooms are then placed by expressing the layout constraints (described above) and then receiving a successful solution from the constraint solver. This places each room at a specific position. Finally, we use the approach described in [1] to connect rooms. First, rooms are connected by computing a Delaunay triangulation [4] among them, followed by computing a minimum spanning tree [15] from this graph. Passageways are used to realize the graph edges, thereby completing the dungeon. Since the focus of this paper is room layout, we omit the mission generation and decoration phases. We note that [1] randomly adds connections from the Delaunay triangulation back into the graph constructed by the minimum spanning tree to increase room connectivity; we omit this step to emphasize control line following.

We are interested in answering these research questions:

R1. How much time does it take to perform dungeon room layout?

Since constraint solvers can take varying amounts of time to compute the solution to a series of equations (or determine they are unsatisfiable), it is important to know how long the solving step takes. In general, this increases with the number of constraints, which increases with the number of rooms. Use of control line constraints increases the total number of constraints, and hence might impact solver time. In general, actual solution times are substantially lower than theoretical maximum times, hence motivating our empirical approach for characterizing performance.

R2. How uniformly are rooms placed within the playfield?

Since the constraints are defining a generative space, the Z3 solver is effectively sampling from this space when it returns each individual solution. Ideally this sampling is occurring in a uniform way, with all solutions equally likely to be returned. If there are hotspots, designers should be aware of this, potentially adding constraints to avoid them.

4.1 Layout Time

To explore time required to perform dungeon layout, we collected data for two main situations: dungeon layout without control line

Table 1: Median times (in seconds) for major operations in dungeon layout, along with counts of the number of constraint clauses

Situation	# And Clauses	# Or Clauses	Setup	Solving	Delaunay	Min. Span Tree	Total Time
Standard 10 room	85	180	0.05157	0.01335	0.001437	0.004464	0.07082
Standard 20 room	270	760	0.1812	0.03557	0.002366	0.0006633	0.2198
Standard 30 room	555	1740	0.4094	0.1262	0.003378	0.0007265	0.5397
Five line 10 room	285	230	0.1158	0.02321	0.001661	0.0007692	0.1414
Five line 20 room	670	860	0.3361	0.07462	0.002610	0.0007077	0.4140
Five line 30 room	1155	1890	0.6203	0.4982	0.003599	0.0007789	1.123

constraints (standard), and dungeon layout where five control lines were present (five line). Representative generated dungeons for each of these situations are presented in Figures 3 and 4. For the standard situation, rooms had widths and heights varying from 20-60 units. For the five line situation, rooms had widths and heights varying from 10-20 units. The smaller size was chosen to ensure a 30 room dungeon was feasible. Runs using 10, 20, and 30 rooms were executed, with a playfield of 400x400 units. In each of the 50 runs, 100 iterations of the same set of room sizes were generated (a total of 5,000 dungeons generated). New room sizes were generated at the start of each run.

Times were collected for each of the major steps in dungeon generation. Experiments were run on a laptop computer with a Core i7 processor (6 cores) running at 2.2Ghz and 32GB of RAM. Code was written in Python, and ran in Ubuntu hosted by Windows Subsystem for Linux (WSL2). Source code and empirical data are available on GitHub[2]. Constraint setup occurs once per run, and encompasses constructing and transmitting all of the constraints to the Z3 SMT solver. Solving is the time required to solve these constraints once (i.e., the time required to perform the layout step). Times required to perform a delaunay triangulation and compute a minimum spanning tree (the most time consuming steps involved in determining passageways) are also reported. Since many frequency distributions are not normal, median values are reported. Table 1 presents timing results, presented in seconds. Figures 5 and 6 presents frequency distributions of solver times (in seconds) for these situations.

Solving time results increase with the number of rooms, and hence the number of constraints. Solving times are higher for the five control line situation, due to the greater number of constraints, and the more limited set of possible solutions at higher rooms numbers. Frequency distributions of solve times for the five-line situation have increasingly long tails as the number of rooms increases, a reflection of the difficulty of finding solutions and exploration of greater parts of the solution space inside the solver.

4.2 Placement Distribution

To explore placement distribution, we collected data from two similar situations: a standard layout with no control lines, and the same five control line situation. As above the standard situation had rooms with sizes varying from 20-60 units in width and height. For the five control line situation, rooms varied from 10-30 units in width and height. 25 runs of 100 dungeon generations were performed for each situation, except the 30 room control line where we used 5 runs of 100 generations, due to an average solver time of 14.43 seconds (this time is greater than those in Fig. 6 due to the larger room size of 10-30 units vs 10-20 units). Heatmaps were computed of the area covered by each room, with a grid size of 5 units. If rooms are uniformly spread out across the playfield, each grid cell should have roughly the same number of rooms covering it across all runs. Any area that has more/less rooms than typical will appear as bright/dark spots in the heatmap, and indicate situations where the constraint solver was unable to uniformly select layouts from the set of potential solutions. Figures 7 and 8 show heatmaps for the standard and five control line situations.

For the standard situation with 10 rooms, room placement is clearly skewed towards the upper left corner. This likely represents the solver quickly finding solutions in this quadrant, and not seeing a need to go further to find solutions. For 20 and 30 room layouts, the placement is far more uniform, with a slight bias towards the upper left and lower right corners. An interesting cross-hatch artifact appears in these heatmaps, of unknown cause. Heatmaps for the five control line situation show a good spread of rooms along the control line, with a slight bias towards the start of each line segment. There is a grid artifact where the rooms are preferentially placed in specific grid locations. This likely represents the fact that room sizes tend to constrain possible solutions, and especially at high room counts, most available valid locations along the control lines need to be used to find a solution.

5 CONCLUSION

We have presented an approach for declaratively specifying dungeon layouts, and have used the Z3 SMT solver to generate dungeons meeting these specifications. This approach permits designers to work in a declarative style, stating what features they desire in a level, which the generator will then satisfy if possible. We have presented two examples of designer constraints, in the form of positional constraints in the throne room example, and the powerful technique of using control lines to guide room placement. Existing dungeon generation approaches lack this degree of designer control.

SMT solvers are a powerful technique for procedural content generation. Many level design problems involve the layout of items within a playfield, subject to various geometric constraints. Industrial strength SMT solvers, such as Z3, make it possible to use declarative level generation for a wide range of level types. We have found Z3 to have good performance for level generation tasks

[2]https://github.com/JimWhiteheadUCSC/smt_dungeon

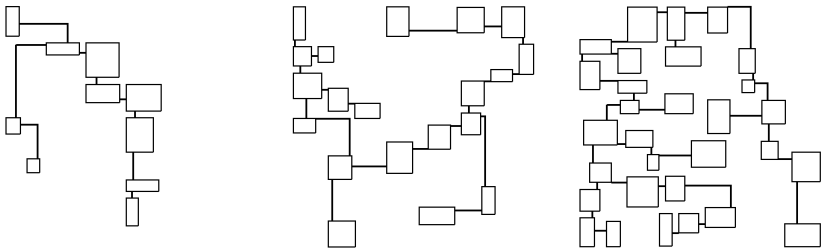

Figure 3: Representative generated dungeons for standard layout with (left to right) 10/20/30 rooms.

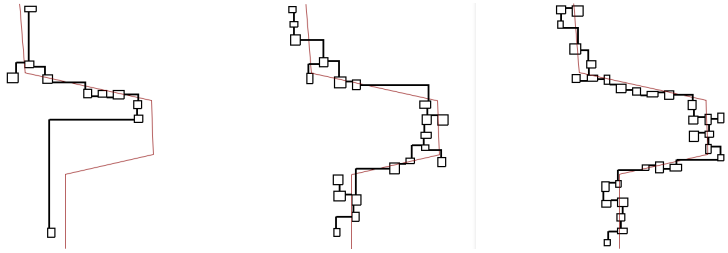

Figure 4: Representative generated dungeons for five control line layout (red lines) with (left to right) 10/20/30 rooms

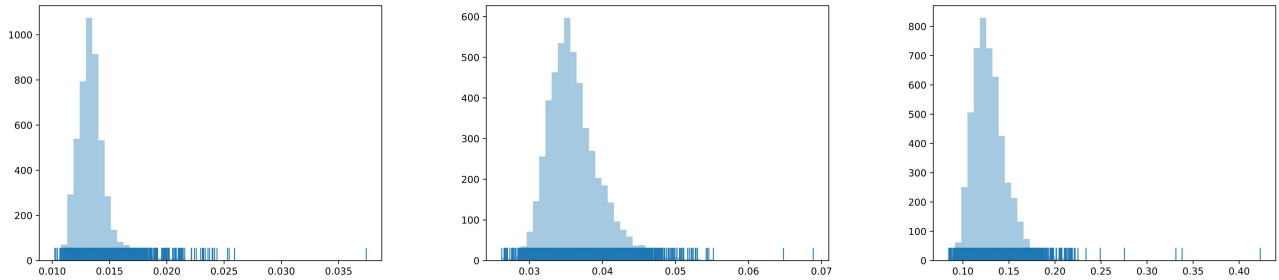

Figure 5: Frequency distribution of solving times for standard layout with (left to right) 10/20/30 rooms. The x-axis is the solving time (in seconds), and the y-axis is a count. Blue ticks (bottom) show individual instances and highlight outliers.

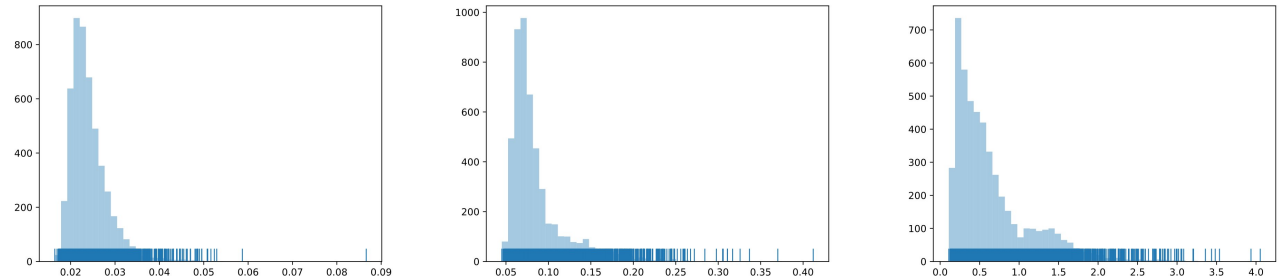

Figure 6: Frequency distribution of solving times for five control line layout with (left to right) 10/20/30 rooms. The x-axis is the solving time (in seconds), and the y-axis is a count. Blue ticks (bottom) show individual instances and highlight outliers.

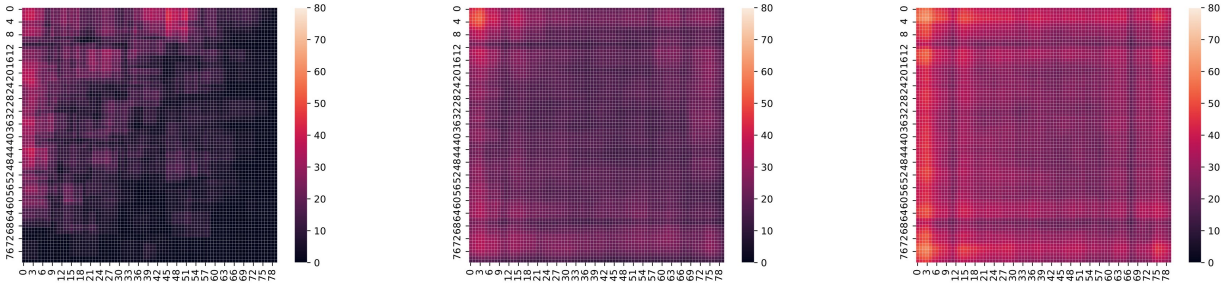

Figure 7: Heatmap of room area coverage for standard layout with (left to right) 10/20/30 rooms

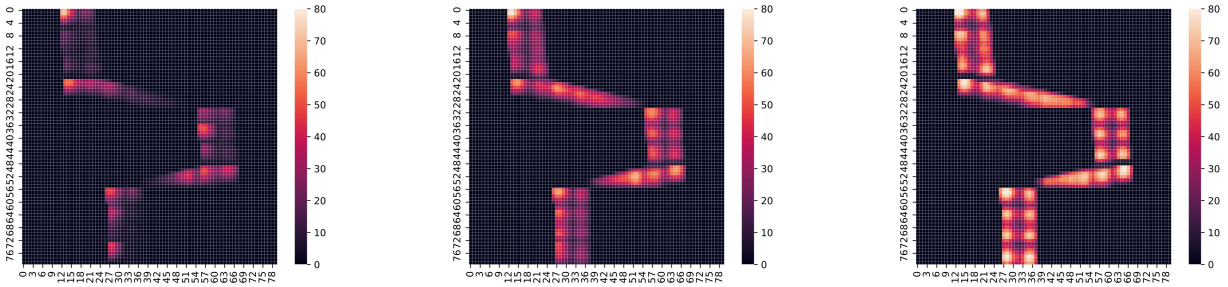

Figure 8: Heatmap of room area coverage for five control line layout with (left to right) 10/20/30 rooms

typical of dungeon crawler type games, and it provides surprisingly good sampling across the space of possible solutions.

In future work, it would be interesting to take advantage of Z3's ability to solve systems comprised of binary and linear constraints to generate missions and spatial layouts at the same time. Further, since a major benefit of the constraint-based approach is the ability to accommodate change, it would be interesting to build a design tool around this linear constraints approach.

REFERENCES

[1] A Adonaac. 2015. Procedural Dungeon Generation Algorithm. https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php Library Catalog: www.gamasutra.com.

[2] Anthony J. Smith and Joanna J. Bryson. 2014. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games. In *Proceedings of the 50th Anniversary Convention of the AISB*.

[3] Antonios Liapis, Giorgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-Aided Game Level Authoring. In *Proceedings of the 8th International Conference on the Foundations of Digital Games (FDG 2013)*. Chania, Crete, Greece.

[4] Franz Aurenhammer, Rolf Klein, and Der-tsai Lee. 2013. *Voronoi Diagrams And Delaunay Triangulations*. World Scientific Publishing Company. Google-Books-ID: cic8DQAAQBAJ.

[5] Alexander Baldwin, Steve Dahlskog, Jose M. Font, and Johan Holmberg. 2017. Mixed-initiative procedural generation of dungeons using game design patterns. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 25–32. https://doi.org/10.1109/CIG.2017.8080411 ISSN: 2325-4289.

[6] J. Timothy Balint and Rafael Bidarra. 2019. A generalized semantic representation for procedural generation of rooms. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG '19)*. Association for Computing Machinery, San Luis Obispo, California, 1–8. https://doi.org/10.1145/3337722.3341848

[7] Jessica R. Baron. 2017. Procedural Dungeon Generation Analysis and Adaptation. In *Proceedings of the SouthEast Conference (ACM SE '17)*. Association for Computing Machinery, Kennesaw, GA, USA, 168–171. https://doi.org/10.1145/3077286.3077566

[8] Bob Nystrom. 2014. Rooms and Mazes: A Procedural Dungeon Generator – journal.stuffwithstuff.com. http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

[9] Carlos Calderon, Marc Cavazza, and Daniel Diaz. 2003. A New Approach to the Interactive Resolution of Configuration Problems in Virtual Environments. In *Third International Symposium on Smart Graphics (SG 2003)*. Heidelberg, Germany.

[10] Daniël Karavolos, Anders Bouwer, and Rafael Bidarra. 2015. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015)*. Asilomar Conference Center, Monterey, California, USA. http://www.fdg2015.org/papers/fdg2015_paper_25.pdf

[11] Danny Lobos and Dirk Donath. 2010. The problem of space layout in architecture: A survey and reflections. *arquiteturarevista* 6, 2 (Dec. 2010), 136–161. https://doi.org/10.4013/arq.2010.62.05

[12] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Budapest, Hungary, 337–340.

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

[14] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. 2006. Persistent Real-time Building Interior Generation. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*.

[15] R.L. Graham and Pavol Hell. 1985. On the History of the Minimum Spanning Tree Problem. *Annals of the History of Computing* 7, 1 (Jan. 1985), 43–57. https://doi.org/10.1109/MAHC.1985.10011 Conference Name: Annals of the History of Computing.

[16] Nathan Hilliard, John Salis, and Hala ELAarag. 2017. Algorithms for procedural dungeon generation. *Journal of Computing Sciences in Colleges* 33, 1 (Oct. 2017), 166–174.

[17] Xavier Ho and Marcus Carter. 2019. Roguelike ancestry network visualisation: insights from the roguelike community. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG '19)*. Association for Computing Machinery, San Luis Obispo, California, 1–9. https://doi.org/10.1145/3337722.3337761

[18] Hoda Homayouni. 2006. *A Survey of Computational Approaches to Space Layout Planning (1965-2000)*. Technical Report. University of Washington, Department of Architecture and Urban Planning. https://www.semanticscholar.org/paper/A-Survey-of-Computational-Approaches-to-Space-Homayouni/3a4cda7185ff22647c8b895117f12c625dae4beb

[19] Ruizhen Hu, Zeyu Huang, Yuhan Tang, Oliver van Kaick, Hao Zhang, and Hui Huang. 2020. Graph2Plan: Learning Floorplan Generation from Layout Graphs. (April 2020). http://arxiv.org/abs/2004.13204 arXiv: 2004.13204.

[20] Jess Martin. 2006. Procedural House Generation: A method for dynamically generating floor plans. In *Proceedings of the Symposium on Interactive Computer Graphics and Games*.

[21] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X. Chang, and Daniel Ritchie. 2019. PlanIT: Planning and Instantiating Indoor Scenes with Relation Graph and Spatial Prior Networks. In *SIGGRAPH*.

[22] Ken Xu, James Stewart, and Eugene Fiume. 2002. Constraint-based Automatic Placement for Scene Composition. *Grahpics Interface* 2 (May 2002), 25–34.

[23] Becky Lavender and Tommy Thompson. 2017. A Generative Grammar Approach for Action-Adventure Map Generation in The Legend of Zelda. University of Sheffield. http://www.t2thompson.com/wp-content/uploads/2016/03/zelda-aisb-2016.pdf

[24] Antonios Liapis, Christoffer Holmgård, Georgios N. Yannakakis, and Julian Togelius. 2015. Procedural Personas as Critics for Dungeon Generation. In *Applications of Evolutionary Computation (Lecture Notes in Computer Science)*, Antonio M. Mora and Giovanni Squillero (Eds.). Springer International Publishing, Cham, 331–343. https://doi.org/10.1007/978-3-319-16549-3_27

[25] Nathan Williams. 2014. *An Investigation in Techniques used to Procedurally Generate Dungeon Structures*. Technical Report. http://www.nathanmwilliams.com/files/AnInvestigationIntoDungeonGeneration.pdf

[26] Charles E. Pfefferkorn. 1975. A heuristic problem solving design system for equipment or furniture layouts. *Commun. ACM* 18, 5 (May 1975), 286–297. https://doi.org/10.1145/360762.360817

[27] Riccardo Lopes, Tim Tutenel, Ruben M. Smelik, Klaas Jan de Kraker, and Rafael Bidarra. 2010. A constrained growth method for procedural floor plan generation. In *Proc. 11th International Conference on Intelligent Games and Simulation (GAME-ON 2010)*. Leicester, United Kingdom, 13–20.

[28] Timothy Roden and Ian Parberry. 2004. From Artistry to Automation: A Structured Methodology for Procedural Content Creation. In *Entertainment Computing – ICEC 2004 (Lecture Notes in Computer Science)*, Matthias Rauterberg (Ed.). Springer, Berlin, Heidelberg, 151–156. https://doi.org/10.1007/978-3-540-28643-1_19

[29] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. 2013. Designing Procedurally Generated Levels. In *Proceedings of the Workshop on AI in the Game Design Process (AIIDE 2013)*.

[30] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, and Rafael Bidarra. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*, Noor Shaker, Julian Togelius, and Mark J. Nelson (Eds.). Springer International Publishing, Cham, 31–55. https://doi.org/10.1007/978-3-319-42716-4_3

[31] Adam M. Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (Sept. 2011), 187–200. https://doi.org/10.1109/TCIAIG.2011.2158545

[32] Thomas Smith, Julian Padget, and Andrew Vidler. 2018. Graph-based generation of action-adventure dungeon levels using answer set programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games (FDG '18)*. Association for Computing Machinery, Malmö, Sweden, 1–10. https://doi.org/10.1145/3235765.3235817

[33] Stanislaus Chaillou. 2019. *AI+ Architecture: Towards a New Approach*. Ph.D. Dissertation. Harvard University, Graduate School of Design. https://www.academia.edu/39599650/AI_Architecture_Towards_a_New_Approach

[34] Petar Borisov Stoykov. 2017. *Rectangle Packing in Practice*. Master's thesis. TU Eindhoven.

[35] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klas Jan de Kraker. 2009. Rule-based layout solving and its application to procedural interior generation. In *CASA Workshop on 3D Advanced Media in Gaming and Simulation*.

[36] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan De Kraker. 2008. The role of semantics in games and simulations. *Computers in Entertainment* 6, 4 (Dec. 2008), 57:1–57:35. https://doi.org/10.1145/1461999.1462009

[37] U. Flemming, R. Coyne, T. Glavin, and M. Rychener. 1988. *A Generative Expert System for the Design of Building Layouts*. Technical Report EDRC-48-08-88. Carnegie Mellon University.

[38] Valtchan Valtchanov and Joseph Alexander Brown. 2012. Evolving dungeon crawler levels with relative placement. In *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering (C3S2E '12)*. Association for Computing Machinery, Montreal, Quebec, Canada, 27–35. https://doi.org/10.1145/2347583.2347587

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

[40] Véronique Gaildrat. 2007. Declarative modelling of virtual environments, overview of issues and applications. In *International Conference on Computer Graphics and Artificial Intelligence (3IA 2007)*. Athens, Greece.