Graph-based Generation of Action-Adventure Dungeon Levels using Answer Set Programming

Thomas Smith
Ninja Theory, ltd.
Westbrook Centre, Milton Road,
Cambridge, CB4 1YG, UK
t.a.e.smith@bath.ac.uk

Julian Padget
Centre for Digital Entertainment,
University of Bath,
Bath, BA2 7AY, UK
masjap@bath.ac.uk

Andrew Vidler
Ninja Theory, ltd.
Westbrook Centre, Milton Road,
Cambridge, CB4 1YG, UK
andrew.vidler@ninjatheory.com

ABSTRACT

The construction of dungeons in typical action-adventure computer games entails composing a complex arrangement of structural and temporal dependencies. It is not simple to generate dungeons with correct lock-and-key structures. In this paper we sketch a controllable approach to building graph-based models of acyclic dungeon levels via declarative constraint solving, that is capable of satisfying a range of hard gameplay and design constraints. We use a quantitative expressive range analysis to characterise the initial output of the system, present an example of the degree to which the output may be altered, and show a comparison with an alternate approach.

CCS CONCEPTS

•**Computing methodologies** → **Logic programming and answer set programming;** •**Theory of computation** → *Abstraction;* •**Applied computing** → *Computer games;*

KEYWORDS

Procedural Content Generation, Generative Methods, Answer Set Programming, Expressive Range

ACM Reference format:
Thomas Smith, Julian Padget, and Andrew Vidler. 2018. Graph-based Generation of Action-Adventure Dungeon Levels using Answer Set Programming. In *Proceedings of FDG, Malmö, Sweden, August 7-10, 2018 (FDG'18),* 10 pages.
DOI: 10.1145/3235765.3235817

1 INTRODUCTION

Since the early days of computer gaming, dungeons and similar spaces within action-adventure games like *Rogue*, *Hack* and their descendants have been procedurally generated, often in highly bespoke and game-specific ways [8, 21]. Several previous approaches to dungeon generation for 2D action-adventure games in academic literature have used graph-rewrite rules and spatial grammars to develop an initial model of the 'mission' within the dungeon (sequence of user actions required for completion) and then further rewrite rules to develop a gameplay space that supports the execution of that mission [7, 8, 12, 13, 22]. However, the presence of *hard playability constraints* (e.g lock and key puzzles) within this domain suggest that a constraint satisfaction approach for procedural generation as described by Smith and Mateas [18] may also be effective. Previous work in this area has successfully demonstrated the use of Answer Set Programming (ASP) for generation of dungeons or similar spaces directly within a tiled grid [14–16]. We propose a combination of the graph and constraint approaches that produces dungeon level models in graph form via constraint satisfaction, by modelling the graph generation problem and associated constraints as an ASP problem formulation and using a domain-independent solver to extract valid models, as in Smith et al. [17].

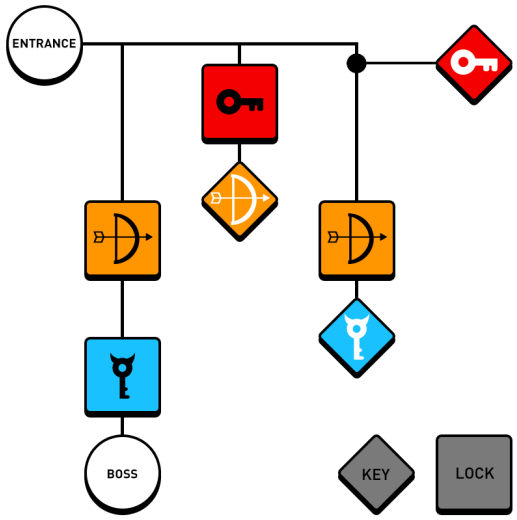

Figure 1: Sample layout of key (diamond) and lock (square) concepts in Boss Key style (Sec. 2.2; image assets from [2]). Concepts are related by colour and icon, connections are traversable paths (abstracting away non-local challenges), and relative vertical positions give a temporal partial order.

To assess the effectiveness of our approach and inform further development we perform an expressive range analysis as described by Smith and Whitehead [19]. This is a quantitative technique for visualising the variety and style of outputs a generator can produce (its 'expressive range'), by calculating and displaying the values of a small number of general metrics over a sample of outputs from the generator. We compare the initial visualisation with the analysis of another generator within the same domain [13] and also with an updated visualisation resulting from minor modifications to the generator constraints, in order to illustrate both the resemblance in outcomes and capability for iteration through varied possible generator outputs.

FDG'18, Malmö, Sweden

DOI: 10.1145/3235765.3235817

Overall we sketch a new approach for producing mission graphs for dungeon levels that differs from the graph-rewriting techniques in literature, and instead builds on Smith and Mateas's work [18], using ASP to generate models of content that fulfill the range of important gameplay and design criteria present in this domain. We demonstrate an application of answer set solving as a generative method for dungeon level models in 'Zelda'-like action-adventure games, and provide a comparison with existing work using an expressive range analysis.

2 BACKGROUND

In this section we introduce the vocabulary for several important dungeon generation concepts; an existing method of visualising dungeons' spatiotemporal structure, and several examples of related work that have informed the current approach.

2.1 Dungeons in action-adventure games

'Dungeons' are a particular kind of contained episodic experience and associated playable area within many action-adventure games. Characterised by a degree of detachment from the main plot of the game (if any) and a complicated physical layout, their self-contained nature, somewhat formulaic structures and typical lack of verisimilitude to any real-world space make them well-suited as candidates for procedural generation.

Common features of dungeons across multiple games include:

— Increasing access: some of the dungeon is initially inaccessible to the player until they have overcome various obstacles or located and used keys/keylike items on locks (or similar concepts [7])

— A variety of obstacles (e.g. combat, puzzles), to avoid repetition

— An element of player choice or exploration, rather than a highly linear sequence of events as in e.g. platform levels [23]

— optional or hidden routes or rewards that make progress through the dungeon easier but are not critical for completion

— Some degree of 'backtracking' or return through previously explored areas: a common trope is having the player first encounter a locked door to indicate that they should search for a key[8]

— A 'Boss': a tougher-than-usual combat or scripted encounter that serves as the final obstacle and goal within the dungeon [10], and whose defeat typically advances some story-related purpose.

Keys and locks form an important part of the hard constraints relating to dungeon playability: in a badly designed or generated dungeon a player might be unable to progress far enough to reach the boss if they cannot find or use the correct keys (see Fig. 2; Sec. 2.2). Dormans [8, pp.91–93] presents a taxonomy of possible key and lock properties that determine the constraints they impose on playability, of which two are presently relevant: keys may be either *consumable* or *persistent* (i.e. able to be used once only or multiple times) and may be *particular* or *non-particular*: able to open a single, specific lock, or any of a class of available locks respectively. In this paper we will be specifically considering constraints patterned on those found in the 'Zelda' franchise of games, which contain three notable classes of keys per dungeon:

— one or more *non-particular consumable* keys (known as Small Keys) and an equal number of associated locks, potentially resulting in a choice of how to progress (see *Limitations*, opposite);

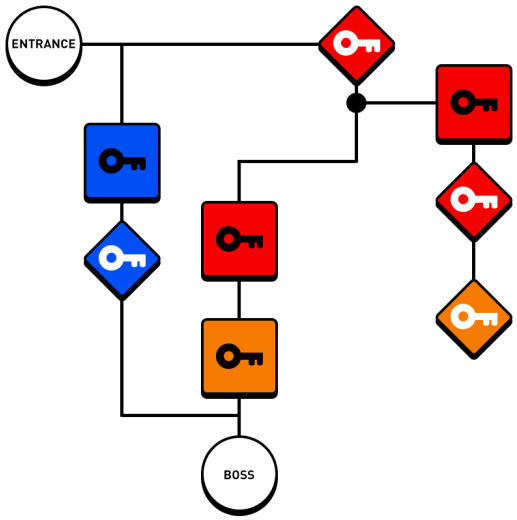

Figure 2: A pair of undesirable challenge arrangements. Left branch: a key is inaccessible behind its associated lock. Right branch: the first red key could be used in the wrong (left) lock, leaving other keys inaccessible (see Sec. 2.2).

— a Boss Lock immediately before the Boss, only unlocked by a *particular consumable* Boss Key elsewhere in the dungeon; and

— a Dungeon Item, a *persistent*, *non-particular* key-like item that allows the player to cross or 'unlock' multiple previously impassable lock-like obstacles blocking the route to the Boss Lock / Boss Key.

Dungeons in the 'Zelda' style contain two kinds of challenges: key-and-lock combinations that relate to the order of progression through the dungeon and require visiting two or more specific locations in a certain sequence, as described above, and also localised challenges that take place entirely within a single small region ('room') of the dungeon. We define such 'local' challenges as those that may be completed using only abilities that the player character may be assumed to possess at the start of any level plus those that are available within the room itself, and they are to a degree interchangeable without affecting the dungeon structure[1]

In contrast, 'non-local' concepts impose constraints on the structure of and critical path through the generated dungeon. They connect physically distant parts of the structure graph via a partial temporal ordering that informs generation decisions (Listings 2-8).

Limitations. The features described in this section each indicate constraints that must be satisfied if we want to generate dungeons in this style, however for this prototype we use modified versions of two of the 'Zelda' restrictions: 1) as in the shape-generation portion of [13] we consider only acyclic dungeon layouts, to assist direct comparison and simplify implementation, and similarly 2) we accept an implementation of the Small Key concept that is *particular*, and therefore potentially provides fewer choices. Many existing Zelda levels are within the space represented by these modified constraints, and so we consider them acceptable for illustrative purposes, however they also represent important future work to be done if increased verisimilitude is desired.

[1]We consider three kinds of local challenge: *combat*, *puzzle* and *traversal*. We presently distinguish these only to ensure a degree of balance and variety among different kinds; however annotation of room nodes with concepts could be useful for future refinement.

2.2 'Boss Key' dungeon graphs

To illustrate these non-local relationships in dungeons within this paper we make use of Mark Brown's 'Boss Key' representation for dungeon lock and key arrangements (Figs. 1, 2 and 4), which was developed to communicate design observations about Zelda dungeon layouts as part of an educational video series [2]. Though it bears similarities to the earlier graph representations used by Dormans [6, 7] and others covered in Sec. 2.3, it is not constructed as an intermediate step for generation but rather as an explanatory tool to address the problems that arise when analysing traditional 2D maps of existing dungeons. It can be unclear from a purely spatial overview of a dungeon which areas are accessible based on changes in player abilities and possessions during the course of the dungeon. The format is therefore designed to omit unnecessary information; clearly map temporal progress in addition to spatial relations, and thereby highlight otherwise difficult-to-read information such as the degree of choice available at different points or the necessity of backtracking through previous areas.

Fig. 1 shows the simplest possible map containing all of the non-local concepts listed previously (actual dungeons can often be significantly more complex; cf. [2]). The lines connecting nodes represent concepts accessible to the player, and their temporal progress through the dungeon is roughly mapped down the vertical direction of the diagram. From the Entrance of the dungeon, one Small Key and three locks are initially accessible (though two locks cannot be opened until later). Once acquired, that Small Key can be consumed to open the matching (central) lock, granting access to the Dungeon Item (in this case, a bow and arrow). The player may then open both Dungeon Item Locks in either order (perhaps by firing arrows at inaccessible levers controlling gateways), providing passage to the Boss Key and the Boss Lock. These may be accessed each in turn to reach the Boss itself, and the end of the dungeon.

In contrast, Fig. 2 illustrates some possible violations of typical dungeon layout constraints. The most obvious is that the arrangement of the blue lock and key on the leftmost branch is reversed, meaning the player is unable to open the lock using the key they cannot access behind it. A more subtle issue is present on the right branch, where it is possible for the player to make choices that leave the dungeon in an incompletable state. If the player chooses to use the red Small Key to open the left red lock rather than the right one, then they are once again in a situation where the key they need next is inaccessible behind the lock it would open on the rightmost branch. Though some games are designed with mechanics that reduce the impact of otherwise incompletable levels [8], in general it is desirable to ensure that these situations cannot arise.

This style of dungeon diagram is a high-level representation of the overall progression of accessible subsections of the dungeon. Notably, it does not include any information about local obstacles within the dungeon, nor does it show any route, key or lock that is not on the critical path between the Entrance and Boss nodes. However, it is useful for illustrating the relationships between non-local challenges, comparing the spatiotemporal structure of two or more visually-dissimilar dungeons, and many of the original dungeons from the Zelda franchise have already been mapped in this style and are available online [2].

2.3 Procedural Dungeon Generation

The simplification afforded by abstracting a dungeon layout as a graph enables a number of possible approaches to generating dungeons using a graph as a starting point. A previous survey on the topic by van der Linden et al. [23] lists a range of techniques and their implementations in literature; many of those most relevant to the present work are described below.

In two early papers on automated top down 2D action-adventure level generation, Dormans [6, 7] details and refines an approach for generating an abstract graph model of both mission and space together, for dungeon-like levels containing locks and keys. The model is transformed via a collection of graph-rewrite systems that successively refined the design model into a functional level.

Van der Linden et al. [22] build on this work with an application to the commercial game Dwarf Quest. They implement a system for constructing designer-guided semantic gameplay grammars, capable of generating action graphs used to generate dungeon levels. They also perform an expressive range analysis (Sec. 4.1) using two bespoke metrics in units particular to their approach.

Karavolos et al. [12] describes a mixed-initiative implementation of Dorman's original concept capable of producing both 2D platformer and dungeon levels, using tile templates and a layout solver to perform the final transformation from model to level, and allowing for direct designer input as part of the generation process.

Separately, Lavender [13] also builds on Dorman's work, presenting a model transformation approach using graph-rewrites and tile templates tailored specifically to the Zelda domain, and discussing the impact of customised sets of rewrite rules designed to produce dungeons exhibiting particular characteristics.

Coming full circle, Dormans [8] introduced a new technique used in the commercial game Unexplored[2] that considers specifically cyclic graphs and transformations that preserve these cycles, due to the congruence between those patterns and common route designs in both the real world and handcrafted levels.

Valtchanov and Brown [21] present an alternative paradigm: evolving a dungeon layout by constructing a population of trees of nodes drawn from a library of templates. They use a fitness heuristic that favours dungeons made up of many small clusters of rooms and Event spaces, connected by hallways.

Baldwin et al. [1] also use an evolutionary approach; to provide diverse offspring of the current map. They use pattern detection algorithms on an edited grid of tiles to identify and construct a hierarchy out of instances of design patterns. Possible successors are selected for according to various design heuristics, and the designer is able to guide mixed-initiative evolution of the level.

Heijne and Bakkes [10] present a system where the main focus of the work is not the generator but the data collection it enables. Their dungeon generation process follows a specific comparatively-simple rules-based approach to assist in comparability of data gathered between runs, but elements of the final reification are adapted at runtime to respond to the skill level of the player.

Finally, Summerville et al. [20] train a range of Bayesian network structures on a corpus of annotated Zelda levels, and show that the networks can learn topology sufficiently for classification. They suggest generation by sampling the network as future work.

[2]Unexplored (PC Game), Ludomotion (22 Feb 2017)

2.4 Answer Set Programming for PCG

Answer Set Programming (ASP) is a declarative logic programming approach aimed at modelling constrained combinatorial search problems. ASP problems are specified by asserting appropriate facts, rules and constraints relating to the domain of interest, and a domain-independent solver is able to return all sets of supported mutually-comprehensible facts that satisfy the assertions.

Smith and Mateas [18] establish an approach for modelling design spaces and thereby formulating content generation problems as ASP logic programs, which result in answer sets that each specify a single instance of valid content. This is achieved by iteratively carving out a desired region of the possible design space using a combination of construction rules and constraints. Smith et al. [17] use this approach to generate an abstract specification for desired puzzles, before embedding the puzzles in a space in a manner similar to that in [7]. Nelson and Smith [14] use the generation of grid-based perfect mazes and simple dungeons as illustrative pedagogical examples in a book chapter on ASP for content generation as constraint-solving problems, including techniques for expressing high-level design goals via universally quantified conditions. Neufeld et al. [15] use a combination of evolutionary algorithms and ASP to evolve solutions for grid-based level generation in the Video Game Description Language domain[3], including dungeons for the VGDL Zelda-inspired game. ASP rules are used to infer the existence and location of objects from the VGDL game definition, with a mutation and evolution approach used to improve numeric parameters. Smith and Bryson [16] describe a system using ASP to assemble room modules from a pool of pre-generated templates into a consistent dungeon layout according to connectivity, and suggest methods for hierarchical refinement of key locations.

Anza Island [4] is an ASP-based puzzle game centred on altering the (re-)generation of a graph of navigable connections, using player-specified constraints such as "Monumental Stone Head can't be connected to Hidden Grotto". The graph generation problem — including player constraints — is formulated in ASP and solved again each turn at runtime to update the game map with valid connections between landmarks.

3 APPROACH

In this section we outline the key elements of our implementation, expand on certain details of the encoding, and present and discuss a sample output from the described formulation (Fig. 3).

Our initial formalization attempts to capture the high-level design concerns and commonalities of the Zelda-like dungeon domain, as described in Sec. 2.1 and including specifically the exceptions relating to acyclicity of the dungeon graph structure, and *particularity* of the Small Keys. We follow the approach laid out in [18] and *construct* a design space through the use of choice rules that generate a selection of available nodes within the graph, *deduce* additional elements of the design space through the use of deduction rules that infer additional necessary nodes, relationships and semantic tagging, and then constrain the design space through the use of integrity constraints that *forbid* undesirable outcomes.

Key elements that are produced by the initial choice rules are the initial pool of nodes and the parenthood relation assignment, which

[3]http://www.gvgai.net/vgdl.php — Accessed 24/05/2018

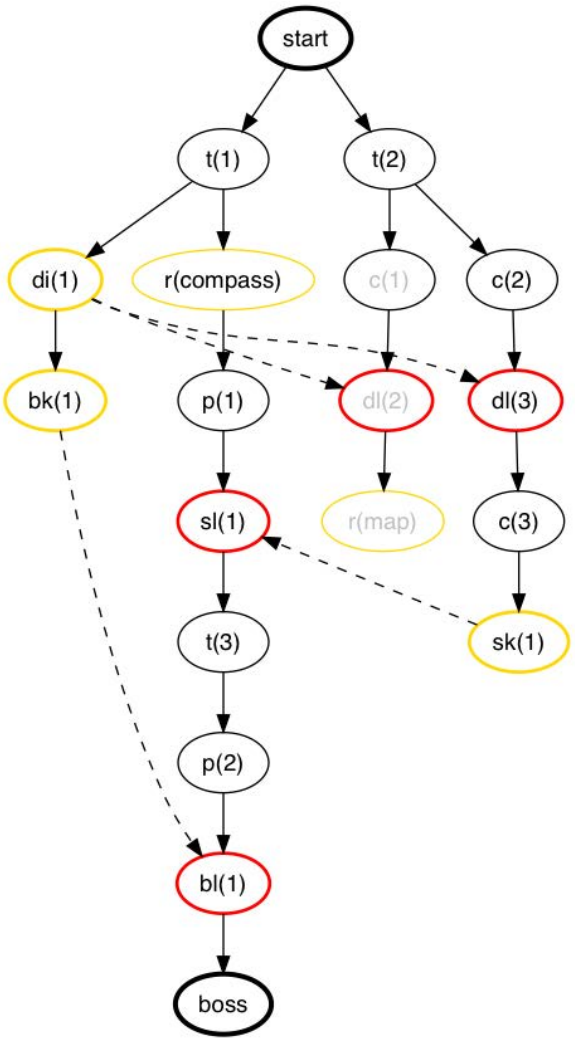

Figure 3: Example of a generated graph with local (black outline) and non-local (bold, coloured red/gold) challenges, rewards (gold), optional path (grey text starting at `c(1)`) and explicit temporal relations (dashed connections). A Boss Key abstraction of this graph is presented in Fig. 4 for comparison, and node IDs are detailed in Table. 1.

together form the basic structure of the graph. Some semantic tags provided by the deduction rules are inherent and listed in Table. 1, such as the quality of being a `keyy(N)` or the `strt(N)`; others are relational and dependent on the assigned parenthood relation. Some integrity constraints represent concerns that are important for gameplay, while others are a matter of designerly intent — in the present formulation they are treated equally.

Within the choice rule for each directly-generated node type are specified an upper bound and lower bound on expected node counts as listed in Table. 1; these initial values were selected observationally based on dungeon mappings in [2] for non-local concepts, and by inspection of existing dungeons for local challenges. The table also specifies the additional tags each node receives, the implications of which are detailed below.

name	ID	min	max	local	tags
Start	start	1	1	*	strt
Combat	c	2	5	*	-
Puzzle	p	2	5	*	-
Traversal	t	2	5	*	-
Small Key	sk	-	-		keyy, rewd
Small Lock	sl	1	3		lock
Dungeon Item	di	1	1		keyy, rewd
Dungeon Lock	dl	2	3		lock
Boss Key	bk	1	1		keyy, rewd
Boss Lock	bl	1	1		lock
Reward	r	2	5	*	rewd, <special>
Boss	boss	1	1	*	rewd, critical

Table 1: Initial configuration of bounds and semantic tags.

3.1 Implementation

To facilitate comparison with existing work in the domain using expressive range analysis, and to simplify both implementation and evaluation we presently consider only dungeons in the form of trees, which may require backtracking but do not contain connections between branches[4]. To efficiently guess a total, acyclic[5] connection, each node is assigned precisely one parent according to the `paft/2` or 'physically after' relation. To begin the process the node labelled `strt` is assigned as its own parent, with a fact stating that `strt` is physically after itself:

```
paft(N,N) :- strt(N).                                          (1)
1{ paft(B, N) : paft(_, B) }1 :- node(N), not strt(N).
```

Thereafter any node `N` that is not labelled as the `strt` is assigned precisely one parent `B` from among atoms that are already a child in a `paft` relationship. This ensures that we generate a single valid tree containing a route through the dungeon and all nodes are ultimately connected to the `strt`. Integrity constraints can be used to disallow undesirable outcomes; for example forbid any answer set where some node has the `boss` or the boss key as its parent, or the `boss` is not behind a lock:

```
%% boss and bosskey must both be terminal                      (2)
:- node(N), paft(boss, N).
:- node(N), paft(bk(1), N).

%% boss must be behind bosslock                                (3)
:- node(boss), not paft(bl(1), boss).
```

A similar approach can be used to tag all nodes that represent dead-ends within the graph (have no known children), and then forbid all answer sets where those nodes are not tagged as a reward — this ensures that the generated dungeon will never contain useless dead ends where a challenge leads to no payoff:

```
terminal(N) :- node(N), not paft(N, _).                        (4)
:- terminal(N), not rewd(N).
```

For non-local concepts there is also a `taft/2` or 'temporally after' relation; represented explicitly in Fig. 3 by the dashed edges connecting keys to locks, and in Fig. 4 implicitly via relationships between vertical heights, and colour-/symbol-coordination. The union of `taft` and `paft` represents a partial order across the nodes in the graph, with `start` and `boss` at first and last respectively.

```
%% trace criticality                                           (5)
critical(boss).
critical(N) :- paft(N, P), critical(P).
critical(N) :- taft(N, T), critical(T).

%% restrict deviation                                          (6)
exploration(N) :- node(N), not critical(N).
:- 5 {exploration(N) : node(N) }.
```

Deduction rules allow us to selectively apply additional semantic tags to nodes, and identify routes that are not on the critical path to the `boss`. These are greyed out in Fig. 3 and faded in Fig. 4 to signify that they are optional — though the constraint in Listing 4 ensures that the optional path will necessarily be rewarding.

```
%% locks imply the existence of their key                      (7)
node(sk(X)) :- node(sl(X)).
taft(sk(X), sl(X)) :- node(sk(X)), node(sl(X)).
keyy(sk(X)) :- node(sk(X)).
lock(sl(X)) :- node(sl(X)).

%% lock cannot be immediately after another lock               (8)
:- paft(X, Y), lock(X), lock(Y).
```

Small Locks, Dungeon Item Locks and the Boss Lock are all part of a `lock/1` category with certain commonalities: e.g. there are never two in a row without some other concept in between; locks are never physically before their own key. Likewise, Small Keys, the Dungeon Item and the Boss Key are all part of the `keyy/1` category.

3.2 Application

This formulation of the dungeon generation problem within ASP occupies 50 lines of code, not counting whitespace or comments. We use Clingo 5.2.1[6] via Python, and configure the solver with `solver.sign_def = "rnd"` and a random seed from numpy. Output from the ASP is a series of facts relating to a model of the dungeon, which the Python script translates into a source format suitable for rendering with GraphViz; one example is shown in Fig. 3, with a Boss Key equivalent in Fig. 4. Several instances of the concepts represented by Listings 1-8 are apparent. Clingo returns a new model in less than one second[7], which facilitates casual experimentation with alternative formulations or varying parameters.

3.3 Reflection

Rendering a single sample output in this way can be instructive while attempting to refine the formulation as it allows easy observation of potentially undesirable outcomes, however without more thorough analysis (covered in Sec. 4.1) it can be difficult to know whether any single flawed production is representative of the generator's typical output. As an example, in the sample output, the leftmost branch contains two key-like items (the Dungeon Item and Boss Key) with no challenges in between. If this is deemed undesirable, there are two possible solutions in Listing 9.

[4] Adding alternate routes, cyclic routes as in [8] and/or shortcuts is left for future work.
[5] mostly. The `strt` node is a special case.

[6] https://potassco.org/clingo — Accessed 06/07/2018
[7] or 10,000 in less than 6s on a 6-core 3.7GHz Windows 10 PC with 16GB RAM.

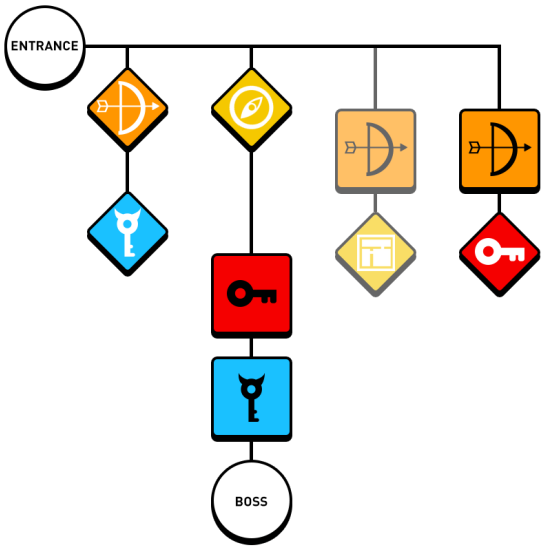

Figure 4: Boss Key abstraction of generated graph in Fig. 3, with only non-local challenges, optional path and rewards.

```
:- paft(A, B), keyy(A), keyy(B).                    (9)
:- paft(A, bk), not local(A).
```

The first rule is modelled on the equivalent formulation for `locks` that forbids any two in a row (Listing 8). The second approach is particular to only the `BossKey` and ensures that it is always preceded by a local concept. Either could fix this specific arrangement, though it is also undesirable to specify too many special-case rules. However, this rapid iteration and refinement of the space of possible outputs is a demonstration of the iterative process proposed by Smith and Mateas [18].

4 EVALUATION

In this section we describe the approach used for evaluation of the systems' output, present a basic comparison with the output of another generator within the same domain, and demonstrate the impact of altering some of the constraints highlighted in the previous section. A key problem with even automated visualisation of output as described in Sec. 3.2 and demonstrated in Fig. 3 is that it is difficult to ascertain how representative a single specimen is. This issue motivated Smith and Whitehead [19] to develop an approach for characterising the 'expressive range' (variety and style) of a generator via visualisation of generator-independent quantitative metrics, sampled over a large number of outputs.

4.1 Expressive Range Analysis

Proposed by Smith and Whitehead [19], expressive range analysis considers the outputs of a generator in aggregate rather than individually. It is an approach that attempts to facilitate interrogation of the generator's range and responsiveness to changed parameters, and can also be used to compare an abstraction over the outputs of two or more generators within similar domains, as in Horn et al. [11] where the technique is used to compare the expressive range of generators in the 2D platforming genre.

Lavender [13] has already made use of the technique within the domain of Zelda-like dungeon generation to analyse her implementation using graph-rewrite rules, which provides a useful point of comparison with an alternative paradigm. We aim to generate comparable heatmaps across the same measures, in order to investigate the expressive range of the present system and compare its performance with another approach.

The expressive range analysis consists of four main steps [19]:

— Determine appropriate metrics. As we intend to contrast the outcome of this analysis with existing visualisations, we will use the same metrics, as defined below. In addition, we measure and report the average size of generated graphs.

— Generate content. We collect 1,000 individual sample dungeon models from separate seeds and collate the metrics scores for each.

— Visualise the generative space. We use matplotlib to render heatmaps of pairs of metrics (Fig. 5), comparable to the existing visualisation by Lavender [13] (Fig. 6).

— Analyse the impact of parameters. In Secs 4.3, 4.4, 4.5 and Fig. 7 we compare and contrast the effects of slight alterations to the problem formulation.

4.2 Appropriate Metrics

We use the following four metrics, as defined by Lavender [13] and based on the original Linearity and Leniency metrics proposed by Smith and Whitehead [19]:

Mission Linearity: the number of nodes on the shortest direct path between start and end of the mission graph, divided by nodes within the graph total. In Fig. 3 this is 9÷19 = 0.473684211.

$$missionL = \frac{Number\ of\ Nodes\ on\ Shortest\ Path}{Total\ Nodes\ in\ Graph}$$

Map Linearity: a weighted scoring of each room with one or more forward exits divided by all rooms with any forward exits: those with a single entrance and exit (fully linear) have weight 1; those with two forward exits have weight 0.5, and those with three exits are considered maximally non-linear and do not contribute to the numerator. 'Dead ends' (rooms with an entrance but no forward exit) are not directly counted by this metric. In Fig. 3 this is $(1 \times 12 + 0.5 \times 3) \div 15 = 0.9$.

$$mapL = \frac{(1{\times}SingleExits) + (0.5{\times}DoubleExits) + (0{\times}TripleExits)}{Total\ Rooms\ with\ Exits}$$

Leniency: the proportion of safe rooms within the dungeon graph to total rooms. For the purposes of this evaluation we have considered only local combat challenges and the final Boss node to be 'unsafe', though the precise calculation of this metric is to a degree dependent on the details of the final realisation of a level: it is possible that any of the traversal or puzzle challenges or even dungeon item locks could be implemented in a way that was potentially 'unsafe' for the player character. In Fig. 3, this is 15 ÷ 19 = 0.789473684.

$$leniency = \frac{Number\ of\ Safe\ Rooms}{Total\ Rooms}$$

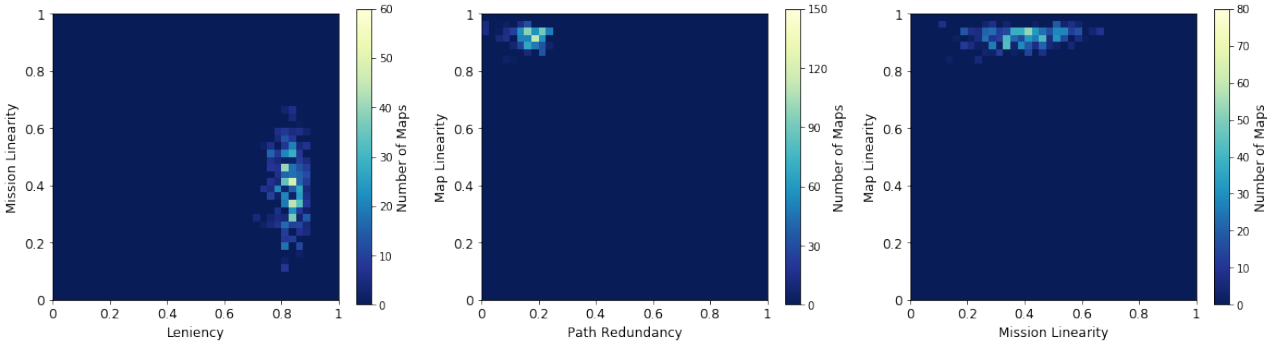

Figure 5: Three views of the expressive range of the initial ASP formulation (Sec. 3.1). This approach to visualisation reveals several potential weaknesses of the initial formulation, as described in Sec. 4.3. Notably, the tight clustering on the Leniency and Map Linearity axes indicates a lack of possible variety in the values of these metrics across all sampled outputs (i.e. generally highly lenient levels, highly linear maps). While it may in fact be desirable for outputs to cluster near these specific values for certain contexts, we wish to show that this generation approach is capable of a broader expressive range. A small number of informed changes to the ASP formulation (Sec. 4.5) results in the considerably more varied output shown in Fig. 7.

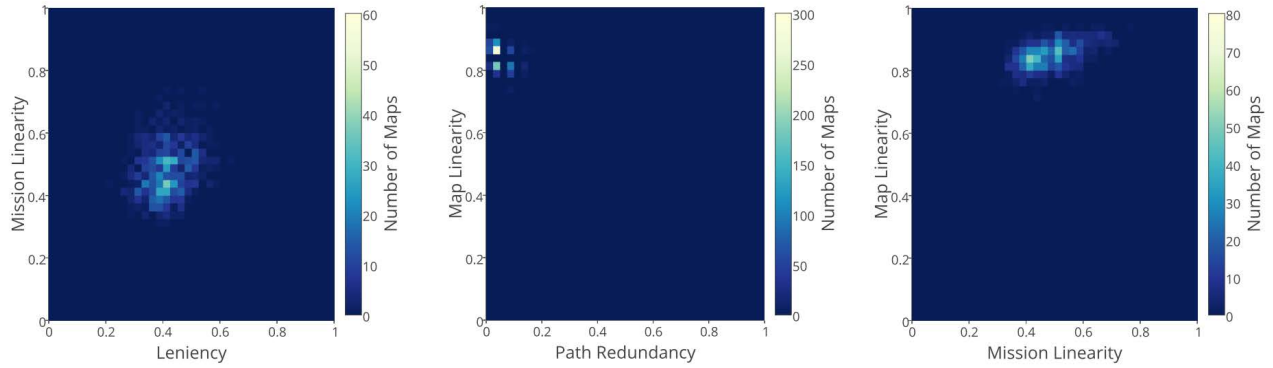

Figure 6: Figures 61, 67 and 70 from Lavender [13], representing evaluation of Control Rules. Reproduced with permission. The approach in Lavender [13] uses a pair of grammars to generate a mission graph and, from it, a mission space. The Mission Linearity and Leniency view (left) is evaluated over the output of the Control mission graph grammar; the Map Linearity and Path Redundancy view (centre) is evaluated over the output of the Control shape grammar over a single mission graph ([13, Fig. 66, p. 78], not reproduced here) and the Map Linearity and Mission Linearity view (right) is evaluated over the output of the Control shape grammar over the output of the Control mission graph grammar. These differing sources explain the gap in Map Linearity at about 0.85 in the centre view, which Lavender suggests may be an artefact of the size of mission chosen.

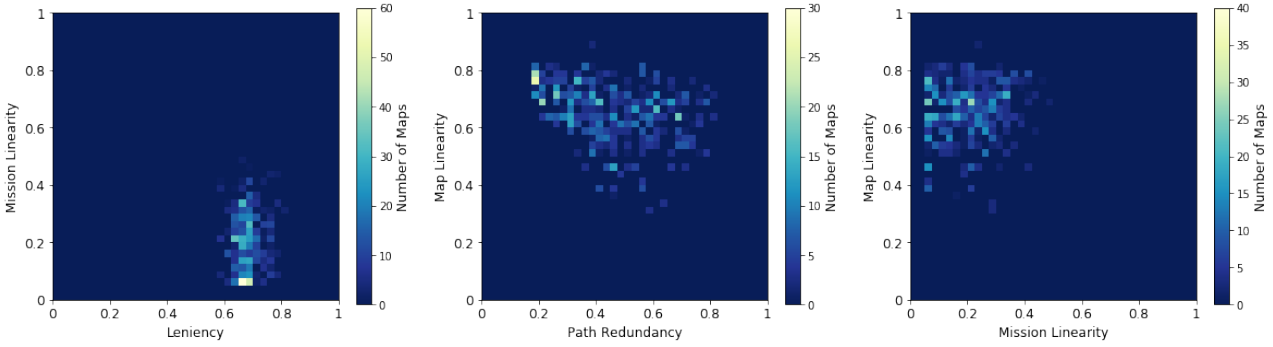

Figure 7: Three views of the expressive range of the altered output generated via ASP after making the changes listed in Sec. 4.5. Note that the variance in Path Redundancy and Map Linearity values has greatly increased compared to the original formulation shown in Fig. 5; the average leniency has decreased; there is increased clustering around the theoretical minimum values for Mission Linearity and Path Redundancy; and the minimum possible value for Path Redundancy appears to be higher. These changes are as expected based on the alterations in Sec. 4.5; informed by this new visualisation additional changes could be made in order to attempt target certain areas of the possible expressive domain if desired, or further broaden the variance.

Path Redundancy: the number of rooms that are present but do not need to be visited in order to complete the level, divided by all rooms. In [13] these are defined as rooms that "do not eventually lead to, or themselves contain, any reward", and are byproducts of possible expansions of the graph-rewriting rules used in that system. However under the ASP formulation described in Sec. 3 these rooms are only generated as optional '`exploration`' paths leading to non-critical `reward` nodes — a comparable but not identical concept. In Fig. 3, these are the combat challenge and dungeon item lock blocking access to the Map reward, and the measure is $3 \div 19 = 0.157894737$.

$$redundancy = \frac{Number\ of\ Non\text{-}critical\ Rooms}{Total\ Rooms}$$

4.3 Initial Results

Having selected appropriate metrics according to the approach laid out in [19] and definitions provided by [13], we generated 1000 dungeon graphs following the approach in Sec. 3, and for each graph calculated the value of the four general metrics. The outcome of this approach is visualised in Fig. 5, where the colour of each square bin in the plot corresponds to the quantity of dungeon graphs possessing those metric values.

Several distinct behaviours of the generator are clear from the visualised heatmap. In general, levels are fairly strongly clustered around a few specific areas, indicating a lack of variety between generator outputs. The leftmost plot shows that all sampled levels are highly lenient, likely due to the low theoretical maximum proportion of dangerous nodes (the maximum possible is five local combat challenges according to the bounds in Table. 1, plus one `boss` node, totalling 6, and the average graph size was 22.363). The second plot shows low path redundancy and high map linearity, likely due to a combination of the rule that forbids dead-ends that don't provide rewards (Listing 4) and the rule that constrains the number of exploration nodes to 5 or less (Listing 6). There are also notable gaps despite the clustering: due to the enforced variety of local challenges and the guaranteed presence of a Boss node, it is impossible for any graph to reach the theoretical maximum Leniency value. Similarly, due to the requirement that the `BossKey` must be terminal (Listing 2), it is not possible for any map to be fully linear.

4.4 Comparison to existing generator

Fig. 6 reproduces three of the outputs of the expressive range analysis performed by Lavender on the system detailed in [13] and summarised in Sec. 2.3 — specifically, the outputs relating to the Control rules: a set of graph- and space-rewrite rules based on those in Dormans [6]. These rules are intended to provide a balance between the other deliberately biased rulesets analysed in that work, and therefore are the most representative point of comparison. The heatmaps reveal a good, central spread of values for leniency and mission linearity, but incredibly tight clustering on the path redundancy metric, apparently due to limitations of the mission graph used. Between the two approaches, the spread of values for both mission and map linearity are reasonably similar, with the primary difference across all four measures being the extreme comparative leniency of the ASP-based levels.

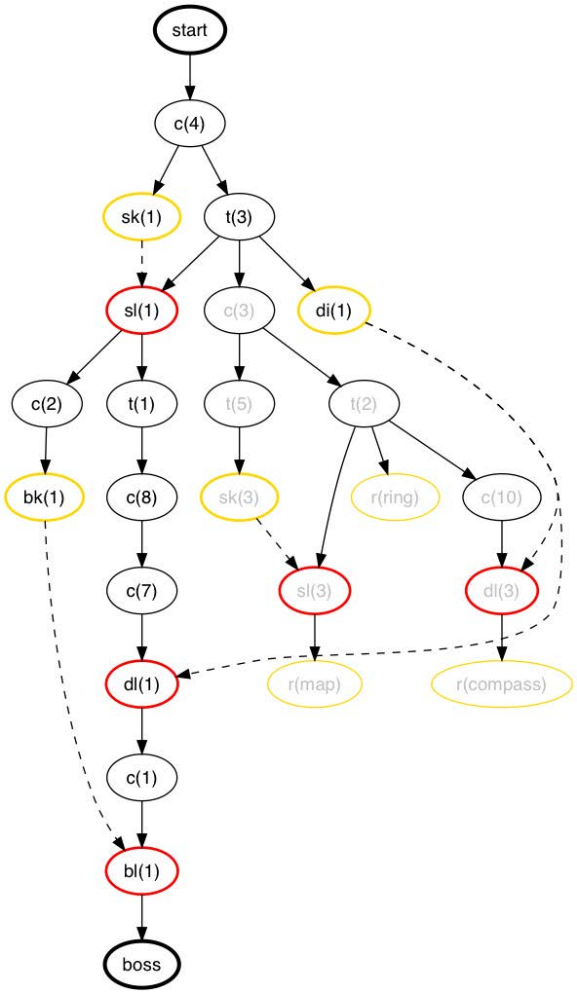

Figure 8: A sample map generated after the changes detailed in Sec. 4.5, showing additional redundancy and nonlinearity.

4.5 Changes to problem formulation

Motivated by the observed clustering in the original visualisation, we investigate the impact of three minor changes to the problem formulation. Working under the hypothesis that the initial path redundancy and map linearity clustering were due to the dead-end and exploration restrictions (Listings 4 and 6), we weaken the former from "`:- terminal(N), not rewd(N).`" to "`:- rewd(N), not terminal(N).`"[8], and we invert the exploration constraint to require a minimum of 5 exploration nodes, rather than 5 maximum. The effects of these changes are clearly visible through comparison of Figs. 5 and 7 — a considerably broader spread. The third change was to replace all potential puzzle nodes with additional combat; resulting in a small but notable decrease in the general leniency. Fig. 8 illustrates a sample dungeon graph generated under the new rules, and clearly shows the effects of constraints requiring increased exploration nodes. As with Fig. 3 in Sec. 4, a single specimen does not indicate how representative of the typical output it is, and so Fig. 9 provides a thorough visualisation of the new space.

[8] 'forbid terminal nodes that are not rewards'→'forbid rewards that are not terminal'

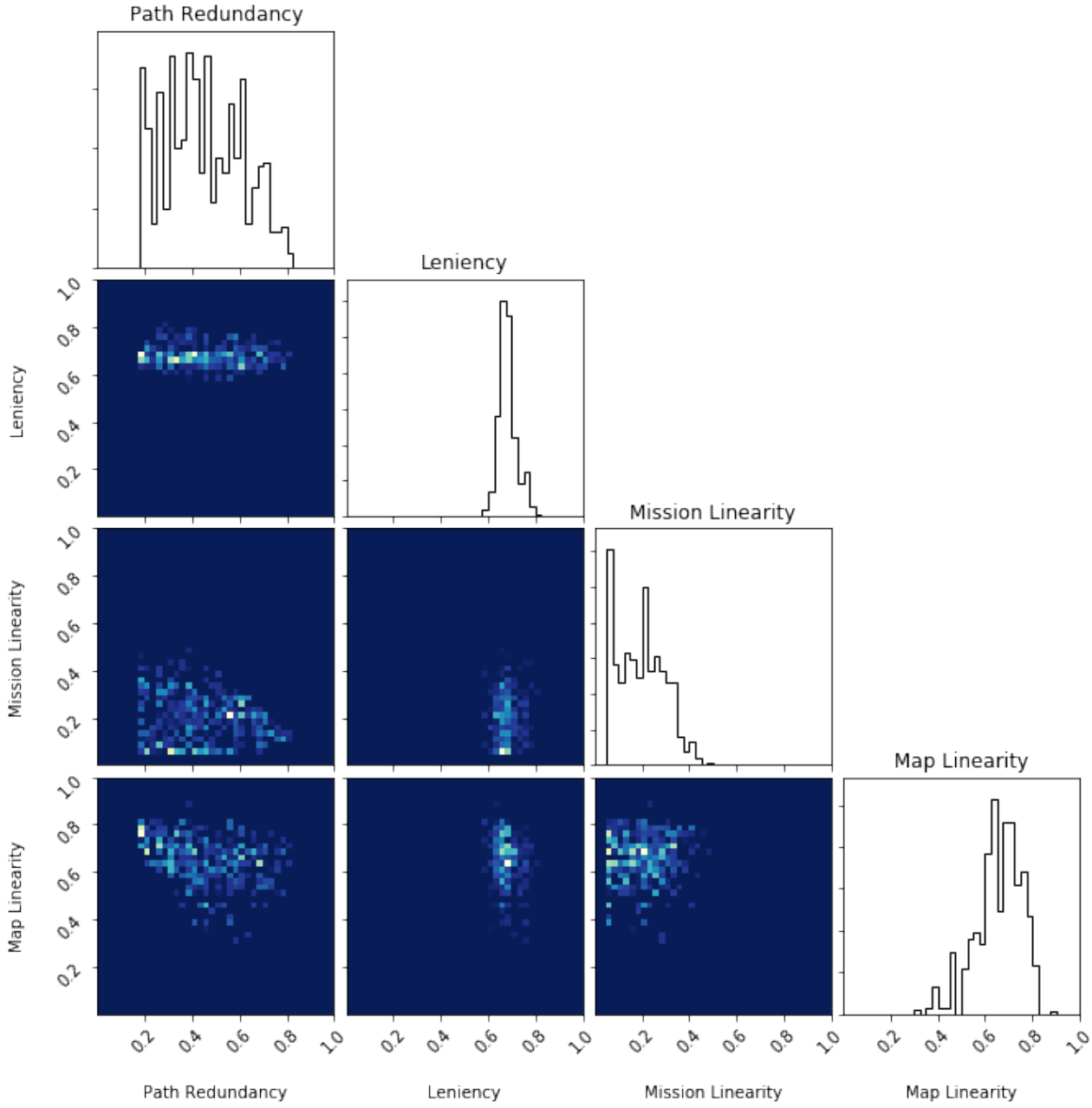

Figure 9: A corner plot [9] showing each of the combinatorial views of the the output data visualisations after making the changes described in Sec. 4.5. Though three of the views duplicate those in Fig. 7 presented for comparison purposes, the remaining three and the single-metric histograms provide additional detail regarding individual distributions. From the Path Redundancy histogram it is apparent that the sampled outputs do not vary smoothly over that metric but rather cluster at a range of specific values, whilst the Leniency variable, viewed in isolation, reveals a continued tight clustering that is more pronounced than is apparent in any of the 2D plots. Visualisations of this nature can help to guide informed changes to the problem formulation, or reveal previously hidden flaws or inexpressible areas within the expressive range [13, 19].

5 FUTURE WORK

The present outcomes suggest a range of promising future work.

An initial improvement would be to increase the sophistication of constraints expressible on the system, such as reasoning over a wider range of concepts present in some traditional Zelda levels including shortcuts between branches or *non-particular* Small Keys. Additional desirable properties might include explicit reasoning over backtracking, the effects of reward items, or layout symmetry. A further step could be to integrate the existing work with a suitable editor such as Solarus [13] or Unreal Engine 4, using Clingo's constraint-solving as the basis of an ASP sculpting interface for dungeon content within the engine. This could also support a more interactive version of the expressive range analysis, akin to the work on Danesh by Cook et al. [5].

Butler et al. [3] propose a method for generating sequences of levels containing an increasing number of concepts. A similar approach could be applied to the present work: if rather than generating levels in isolation a sequence of levels are generated according to a generated progression specification, additional functionalities are possible. Notably, Dungeon Item Locks relating to Dungeon Items that are known to have been collected in prior dungeons are then potentially available for use as local challenges (see Sec. 2.1).

There is also potential for a more in-depth comparison of the generative space characteristics of a wider range of generators within this domain, along with investigation into altering parameters, grammars or constraint formulations. A study by Horn et al. [11] in the domain of 2D platformer (Mario) levels considered seven generators from literature and levels from the original Super Mario Bros game. There are a comparable number of dungeon generators in literature [1, 7, 10, 12, 13, 16, 21, 22, present paper], though this domain lacks the broad consensus on common assumptions and definitions the domain of Mario generators has achieved.

6 CONCLUSION

In this paper we present a work-in-progress approach for generating graph models of action-adventure dungeons using a declarative constraint-based formulation and an off-the-shelf solver. We apply a quantitative analysis in order to investigate the expressive range of the initial formulation, and compare this system to a previous grammar-based generation approach for similar content. We note that using ASP enables us to easily carve out desired areas of the generative space whilst also continuing to satisfy hard gameplay- or implementation-related constraints.

7 ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the Engineering and Physical Sciences Research Council (EPSRC) through the Centre for Digital Entertainment, under grant reference EP/G037736/1. We would also like to thank Becky Lavender for permission to reproduce figures from her work as a point of comparison (Lavender [13], Fig. 6), and Mark Brown for developing the Boss Key dungeon graphing approach and making the visual resources freely available (Brown [2], used in Figs. 1, 2 and 4). We also thank our anonymous reviewers, in particular for the recommendation of corner plots (Foreman-Mackey [9], Fig. 9) as an appropriate visualisation for the expressive range metrics across more than two dimensions.

REFERENCES

[1] Alexander Baldwin, Steve Dahlskog, Jose M. Font, and Johan Holmberg. 2017. Towards Pattern-based Mixed-initiative Dungeon Generation. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17)*. ACM, New York, NY, USA, Article 74, 10 pages. DOI: http://dx.doi.org/10.1145/3102071.3110572

[2] Mark Brown. 2017. How my Boss Key dungeon graphs work. (August 2017). https://www.patreon.com/posts/how-my-boss-key-13801754 Accessed: 2018-06-29.

[3] Eric Butler, Adam M Smith, Yun-En Liu, and Zoran Popovic. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 377–386.

[4] Kate Compton, Adam Smith, and Michael Mateas. 2012. Anza island: Novel gameplay using ASP. In *Proceedings of the The third workshop on Procedural Content Generation in Games*. ACM, 13.

[5] Michael Cook, Jeremy Gow, and Simon Colton. 2016. Danesh: Helping bridge the gap between procedural generators and their output. In *Proceedings of the 7th International Workshop on Procedural Content Generation in Games*. ACM.

[6] Joris Dormans. 2010. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*. ACM, 1.

[7] Joris Dormans. 2011. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*. ACM.

[8] Joris Dormans. 2017. Cyclic Generation. In *Procedural Generation in Game Design*. CRC Press, 83–96.

[9] Daniel Foreman-Mackey. 2016. corner.py: Scatterplot matrices in Python. *The Journal of Open Source Software* 24 (2016). DOI: http://dx.doi.org/10.21105/joss.00024

[10] Norbert Heijne and Sander Bakkes. 2017. Procedural Zelda: A PCG Environment for Player Experience Research. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG '17)*. ACM, New York, NY, USA, Article 11, 10 pages. DOI: http://dx.doi.org/10.1145/3102071.3102091

[11] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. 2014. A comparative evaluation of procedural level generators in the mario ai framework. (2014).

[12] Daniël Karavolos, Anders Bouwer, and Rafael Bidarra. 2015. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation.. In *FDG*.

[13] Rebecca Lavender. 2016. The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games. (2016).

[14] Mark J Nelson and Adam M Smith. 2016. ASP with applications to mazes and levels. In *Procedural Content Generation in Games*. Springer, 143–157.

[15] Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. 2015. Procedural level generation with Answer Set Programming for General Video Game playing. In *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*. IEEE, 207–212.

[16] 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*.

[17] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 2012. 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 (FDG '12)*. ACM, New York, NY, USA, 156–163. DOI: http://dx.doi.org/10.1145/2282338.2282370

[18] 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 (2011), 187–200.

[19] Gillian Smith and Jim Whitehead. 2010. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM.

[20] Adam J Summerville, Morteza Behrooz, Michael Mateas, and Arnav Jhala. 2015. The learning of zelda: Data-driven learning of level topology. In *Proceedings of the FDG workshop on Procedural Content Generation in Games*.

[21] 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*. ACM, 27–35.

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

[23] 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 (2014), 78–89.