A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games

Anthony J. Smith [1] and **Joanna J. Bryson** [2]

Abstract. The development of procedural content generation (PCG) methods for video games is an established area of research. There are many approaches to the problem that utilise a variety of techniques from different fields of computer science. The use of Answer Set Programming (ASP) for PCG in video games is relatively new, however recent research has demonstrated valuable aspects of ASP in the generation and evaluation of design spaces.

This research takes the good work already achieved using ASP for PCG and progresses it to investigate the open issue of scalability. The genre of *Roguelike* games provides the design space of sufficient size and complexity to investigate this scalability issue. Preliminary findings indicate that ASP is a viable option for PCG in video games, in particular demonstrating that a hierarchical application of this technology can deal with such complex game environments.

1 INTRODUCTION

PCG has been used in the games industry for over three decades, to assist level designers by automatically, or semi-automatically, generating game content. A variety of approaches are employed that range from pseudo-random number generators and fractal algorithms, to multi-agents and genetic algorithms. Each approach has its own strengths and weaknesses, resulting in a diverse and disparate range of methods.

One approach that has recently emerged is the use of Answer Set Programming (ASP) with the AnsProlog language, which has shown much promise. The *Chromatic Maze* [13] illustrates how the non-monotonic nature of ASP is useful for procedural content generation by allowing the user to refine the game space by iteratively adding constraints to the AnsProlog program. The *Refraction* game [11] demonstrates the ability of AnsProlog to identify and remove unwanted solutions, in this case puzzles that have shortcut solutions, a valuable capability for procedural content generation. The *Variations Forever* game [12] relates how AnsProlog can be used to define game rulesets as well as game content, indicating the versatility of this approach.

ASP has already shown much potential for generating procedural content, however the open issues of scalability will be considered in our present research. The scalability issue relates to how well ASP can scale up to more complex game environments. ASP is a search-based technique where the AnsProlog program defines and refines a search space of solutions. For simple puzzle games this search space will be relatively small, however for more complex games the search space can dramatically increase, leading to an exponential increase in the PCG execution time. For example, Smith et al. (2013) report a solving time in excess of 2 minutes for an extreme case of their 10x10 puzzle game *Refraction*[11].

The main contribution of our research will be to explore the issue of scalability by using ASP for PCG for *Roguelike* games. We propose a hierarchical approach comprising two phases to PCG, the first phase providing the level structure, the second phase providing the level content. Here we outline in detail the first phase of this hierarchical PCG method since this phase is critical to addressing the scalability issue.

We have outlined the case for using ASP for PCG in games, identified the open issues of scalability, and identified *Roguelike* games as a vehicle for this research. The remainder of this document will provide a concise survey of the current technologies and methods applicable to our research, outline the method and implementation of the first phase of PCG, present and discuss our preliminary results, and suggest future work.

2 BACKGROUND

The aim of this research is to explore the opportunities provided by ASP for generating game level data. Of primary concern is how well ASP can deal with complex game spaces. This scalability issue will be explored by implementing PCG for *Roguelike* games. Here we survey contemporary PCG methods used in games, take an overview of ASP and how it has been used for PCG to date, and then review *Roguelike* games and the PCG methods they tend to employ.

2.1 Procedural Content Generation

Procedural content generation has been used in the video games industry since the late 1970s where it was pioneered in games such as *Rogue* and *Elite*. There is currently no uniform approach to the problem, however there exist a number of methods from different areas of computer science that can be seen as a *toolbox* for PCG.

A standard PCG method is to randomly create content and distribute that content randomly within the design space. The use of Pseudo-Random Number Generation is one such approach where the data is generated using a seeded algorithm allowing the process to be deterministic. This is an important feature since the automatically generated data can be consistently reproduced and therefore

[1] University of Bath, UK, email: a.j.smith@bath.ac.uk
[2] University of Bath, UK

tested. Perlin Noise is a pseudo-random number technique developed to make computer generated images look more realistic [10]. This technique can be applied to a height map to create suitable terrain details such as water, cliffs and plains, or it can be applied to game objects prior to rendering to give them a more realistic (less homogenous) appearance. Fractal algorithms are used to create vegetation and realistic landscapes, these algorithms tend to require very little code to implement, however they can be expensive in processing time.

A variety of methods from AI have been used for PCG including genetic algorithms, neural networks, agent based simulation, constraint satisfaction and search. For example, agent base simulation has been used to manipulate a basic terrain by a set of prescribed software agents in a controlled and specified manner to create realistic landscapes [2].

A common approach for PCG is to divide the game space into a number of useful areas, thus providing structure to the game space, and then populate these areas with content. Johnson et al. [5] used cellular automata for such an approach to generate infinite cave levels, they claim that their method provides a good level of control over the resulting design space, thus tackling one of the open issues with PCG. A similar approach is used in architecture to model towns and cities by dividing the space into regions with a road network, and then populating those regions with buildings. For instance *Instant Architecture* [15], and *CityGen* [6] use generative grammars for both organic and structural generation of content. Developed to understand the rules of language, generative grammars use a set of symbols that are modified by a set of rewriting rules that are applied to the symbols to modify and replicate them.

L-systems are a fractal like grammar that has been used to grow road networks within a design space [9], providing the basic structure of the cities being created. Tensor fields is another method that has been used to generate road networks as an initial process for generating cities[1]. L-systems can further be used to generate the buildings to populate the city [9], however other generative grammars such as split-grammars, wall-grammars, and shape-grammars which are more suited to generating structured components, have also been used. Split-grammars are similar to L-systems, however they manipulate encoded shapes rather than strings (as with L-systems). The process works by using the rewriting rules to convert shapes into new shapes, the process being repeated until the desired shapes are produced [15]. Shape-grammars are like split-grammars in that they perform rewriting rules on encoded shapes, however the rewriting process is influenced by the neighbours of the shape being processed. This allows shape-grammars to produce more complex structures than split-grammars [8]. Wall-grammars are used to generate building facades from shapes that are manipulated by a rewriting process to extrude flat shapes into the third dimension [7]. These approaches produce good results that can be manipulated in real-time, however their genotype to phenotype mapping is weak meaning that the resultant cityscape lacks the semantic meaning needed for a game environment. However, the ability for these methods to produce high quality structures could be very useful in game genres such as *Roguelike*.

2.2 Answer Set Programming

One approach that has recently been adopted for procedural content generation is to use Answer Set Programming (ASP). ASP is a form of logic programming that falls into the declarative programming paradigm; where the program describes the requirements for the solution to a certain problem [3, p. 40]. The program, written in AnsProlog, comprises a set of rules and predicates, that are run through a solver to produce a list of all the possible solutions, referred to as the answer sets.

Following sound software engineering practices an AnsProlog program will be coded in four distinct sections. The *define* section is where all the known facts and rules about the system are encoded. The *generate* section is where derived facts about the system are generated. The *test* section is where integrity constraints are used to remove unwanted solutions. The *evaluate* section is where a stepwise assessment of the system is conducted.

Typically the initial state of the problem is defined by a set of facts that represent the known conditions of the problem space. A set of rules are then defined to represent all the legal changes to the problem space that are allowed. Finally a set of constraints are applied to the problem space, these identify all the illegal states that the system must avoid.

The AnsProlog language realises these constructs with *rules*, *facts*, and *integrity constraints*. A *rule* is defined as $head$ *:-* $body$. where the *head* represents the rule being defined, and the body defines the logic components that satisfy the rule. A *fact* is defined as $head$. it represents something that is known unconditionally, it is a *rule* without a *body*. An *integrity constraint* is defined as *:-* $body$. it represents all the things that are known not to be, it is a *rule* without a *head*. A further useful AnsProlog construct is the choice rule, represented as $l\{rule\}u$, this provides a convenient way of choosing randomly a number of rules from the set of valid rules, the number of which is in the range l and u.

The *Variations Forever* game is a set of mini-games that have procedurally generated rules [12]. This game illustrate how ASP can be used for PCG demonstrating the generate-and-test approach for PCG development, as well as using ASP to produce both game content and game mechanics. The *Chromatic Maze* is a game that automatically generates a maze puzzle represented by a grid of coloured tiles, where the player must find a valid route from the start location to the finish [13]. This simple game illustrate the non-monotonic nature of ASP that allows for abductive reasoning, thus enabling the PCG to be adapted by the level designer. *Refraction* is an educational game that comprises a tiled game space where beam splitters, combiners and benders are placed to manipulate laser beams between emitters and receptors [11]. This game illustrates how ASP can be used to remove unwanted solutions from the set of possible solutions, in this case those puzzles that have unwanted short cuts. A dungeon map generator for a fictional *Roguelike* game uses a hybrid of ASP and evolution computation for PCG [14]. The genetic algorithm is used to generate parameters for the ASP program, the resulting level map is then evaluated against prescribed metrics.

2.3 Roguelike Games

Roguelike games are based on the dungeon questing games such as *Rogue* (1980) and *Moria* (1983). These games where originally designed for computers that had limited or no graphics capabilities, hence they used ASCII characters positioned on the 2D grid that made up the screens display space. Procedural content generation has been used in *Roguelike* games since their inception, a concept that has been widely adopted in most *Roguelike* games to the present day

Roguelike game have a typical set of attributes that are common across the genre. The game space will comprise a 2D grid of navigable tiles, such that the environment is discrete rather than continu-

ous. Each level will comprise a number of rooms, of varying size and shape, connected by corridors that will be positioned such that neither the rooms or the corridors overlap or crossover. The rooms will be populated by a variety of monsters, treasure, equipment, weapons, and traps. The generation of level content will be driven by factors such as the current level difficulty, the chosen game difficulty, and the current level of the player's character, to produce levels that are progressively challenging for the player.

A traditional approach for *Roguelike* games is to use a pseudo-random technique to generate content, for example *NetHack* uses random numbers to generate its dungeons and creatures. The game space is easily represented as a 2D grid of tiles that depict the discrete navigable positions within the game. Pseudo-random number generation (PRNG) techniques can then be used to randomly create content and randomly place the content within the game space. This approach has the benefit of being deterministic, allowing it to be used at run-time to generate known level content in real-time. Fractal algorithms have also been used in *Roguelike* games, for instance *Dwarf Fortress* use fractals to generate elevation maps and other environment features that are combined into a world map.

3 METHOD

The main focus of our present research is to assess whether ASP is a good candidate for generating procedural content for large design spaces (in the order of 100 times the size of the Chromatic Maze). For this purpose, procedural content will be generated for a *Roguelike* game.

3.1 System Overview

Here we describe the system that has been developed to achieve this. The system comprises a python program and some AnsProlog code, that in conjunction take input parameters from the user (level designer) and generate a file containing the level data. This level data file can then be used in the *Roguelike* game *Dungeon Crawl Stone Soup* by copying it to the appropriate directory in the game installation.

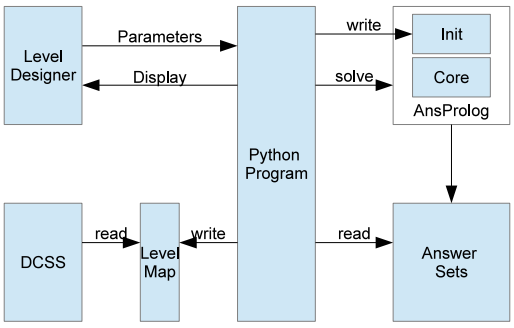

Figure 1. System Overview: The python program takes input parameters from the level designer, these are encoded as logic rules in the AnsProlog program init section. The python program then uses *clingo* to solve the AnsProlog program that produces the answer sets. The python program then interprets the answer sets into a dungeon level map.

3.1.1 Python Program

The main program is written in python. It provides the user with the facility to set a number of parameters to control the level generation. From these parameters it generates the AnsProlog code for initialising the procedural content generation. It then uses the *clingo* grounder/solver to solve the AnsProlog code, which generate the answer sets. It then reads back the answer set to interpret the level content to generate, this is written to an ASCII text file in the format appropriate for the chosen game.

3.1.2 AnsProlog

Answer Set Programming will be used to implement the procedural content generation. The core of this system will be an AnsProlog program. This program will generate the game level data as answer sets, based on a given set of parameters.

The AnsProlog code will be split across two files; one file contains the core of the procedural content generation code, the other file contains initialisation code. The initialisation code will be created by the python program based on the user defined input parameters. This file will contain definitions for parameterised values such as the number of regions to use, and lists of data such as types of monsters to use in the level. The following code segment illustrates the content of this initialisation file.

```
#const number_of_regions = 6.
#const region_size = 10.
monster_types(rat).
monster_types(bat).
monster_types(kobold).
monster_types(orc).
monster_types(zombie).
monster_types(spectre).
monster_types(hydra).
```

The core code has been developed to perform the PCG, this file is manually coded and does not change during or between invocations of the system. Examples of the content of this file is shown in the section 3.4.

The python program invokes the AnsProlog by making a system call to the *clingo* tool. *Clingo* is an Answer Set Programming grounder/solver developed by Potsdam University, further information and the ASP tools can be found at http://potassco.sourceforge.net. The tool is invoke using the following command:-

```
$ clingo init.pl core.pl
```

3.1.3 Dungeon Crawl Stone Soup

The ASCII text files generated can be visually checked by the designer to ensure that the level generated looks right. Inaccessable rooms, unconnected passage ways and poorly distributed content will be easily apparent, however it is more difficult to assess the playability of a level.

Dungeon Crawl Stone Soup is an open source *Roguelike* game that provides some facility of importing game levels. DCSS has an official website at http://crawl.develz.org/wordpress/. The python program generates the level content in the format appropriate for this game, and example of which is shown in figure 2. The level content file is simply copied to the appropriate directory in the game installation where it is picked up by the game. The ability to play the level will provide much better feedback regarding how playable the level is.

```
NAME:  ASP Generated Map 1
MONS:  rat, bat, kobold, orc, zombie, hydra
DEPTH: 1
MAP
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx.....xxxxxxxxxxxxxxxx.....x
xxxxxxxxxxxxxx.....xxxxxxxxxxxxxxxx.....x
xxxxxxxxxxxxxx.......xxxxxxxxxxxxxx......x
xxxxxxxxxxxxxx.....xxxxxxxxxxxxxxxx.....x
xxxxxxxxxxxxxx.....xxxxxxxxxxxxxxxx.....x
xxxxxxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
x.....xxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
x.....xxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
x................xxxxxxxxxxxxxxxxxxx.xxx
x.....xxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
x.....xxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.....................xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxxxxxxxx.xxxxxxxxxxxxxxxxxxx.xxx
xxx.xxxxxx.xxxxxx.xxxxxxxxxxxx.xxxxxx.xxx
xxx.xxxx.....xxxx.xxxxxxxxxxx...xxxxx.xxx
xxx.xxxx.....xxxx.xxxxxxxxxx.....xxxx.xxx
xxx.xxx..........xxxxxx..........xxx.xxx
xxx.xxxx.....xxxxxxxxxx.xxxx.....xxxx.xxx
xxx.xxxx.....xxxxxxxxxx.xxxxx...xxxxx.xxx
xxx.xxxxxx.xxxxxxxxxxx.xxxxxx.xxxxxx.xxx
xxx.xxxxxxxxxxxxxxxxxx.xxxxxx.xxxxxx.xxx
xxx.xxxxxxxxxxxxxxxxx.......xxx.xxx.....x
xxx.xxxxxxxxxxxxxxxx..x.x..xxx.xxx.....x
xxx........xxxxxxxxx.....................x
xxx.xxxxxx.xxxxxxxxx..x.x..xxx.xxx.....x
xxx.xxxxxx.xxxxxxxxx.......xxx.xxx.....x
xxx.xxxxxx.xxxxxxxxxxxx.xxxxxx.xxxxxx.xxx
xxx.xxxxxx.xxxxxxxxxxxxxxxxxx.xxxxxxxxxx
x......xxx.xxxxxxxxxxxxxxxx.......xxxxxxx
x......xxx.xxxxxxxxxxxxxxxx.......xxxxxxx
x.................................xxxxxxx
x......xxxxxxxxxxxxxxxxxxx.......xxxxxxx
x......xxxxxxxxxxxxxxxxxxx.......xxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
ENDMAP
```

Figure 2. Example Level File for Dungeon Crawl Stone Soup. Here the design space is a 42-by-42 grid of ASCII characters where x and . represents wall and floor section respectively. The design space was divided into 6-by-6 regions where each region was assigned a room or passage section.

3.2 Investigating the Scalability Issue

ASP is able to solve NP-hard problems, therefore it makes a good candidate for scaling well as the problem complexity increases [4]. However, generating procedural content for games is often required to be performed online in real-time, or at least to be responsive enough for an interactive design tool. From the recent work on the *Chromatic Maze* [13], and *Refraction* [11], we know that relatively small design spaces can be expensive in terms of time to evaluate. Here we are generating a two-dimensional dungeon map that is made up of 60-by-60 tiles, this is 100 times the size of the *Chromatic Maze* and 36 times the size of the *Refraction* design spaces.

3.2.1 *Distinguishing between Structure and Content*

A key idea in this research towards generating procedural content is to realise that some of the content generated is structural, and the rest of the content are artefacts. Structural components define the environment of the design space, in effect the level map. These structural components are items that do not tend to move, and cannot be picked up, these include walls, floors, water, stairs, and doors. Artefacts are the other components of a typical *Roguelike* dungeon such as monsters, weapons, food, and gold. These artefacts can move, can be moved, or can be picked up.

The distinction is important, since the generation and placement of these artefacts is dependent on the size, shape and placement of the structural components. For instance it would be unusual to place a monster within a wall section. Therefore the process of procedural content generation will generate the structure of the dungeon (the level environment) and then generate the artefacts that populate the dungeon (the level contents).

3.2.2 *Using the Chromatic Maze as a Benchmark*

The Chromatic Maze puzzle [13] can be used as a good benchmark. Here a six by six grid of coloured tiles represents a maze, where movement is only allowed to adjacent tiles that have the permitted colour. The colours used to generate the maze can be thought as forming a colour wheel. Permitted movements around the maze are restricted to adjacent tiles with the same colour or the colour of the two neighbouring colours in the imagined colour wheel.

Using six colours for the maze Smith & Mateas claim a generation time of 2.5 seconds, for a 35 step (optimal solution) puzzle that utilises every location in the maze. This increases to two hours for a maze that has a 21-by-21 grid of coloured tiles, for a sub-optimal solution using 114 steps. Figure 3 shows an example of the Chromatic Maze, with solution illustrated by arrows, and the imagined colour wheel for clarity.

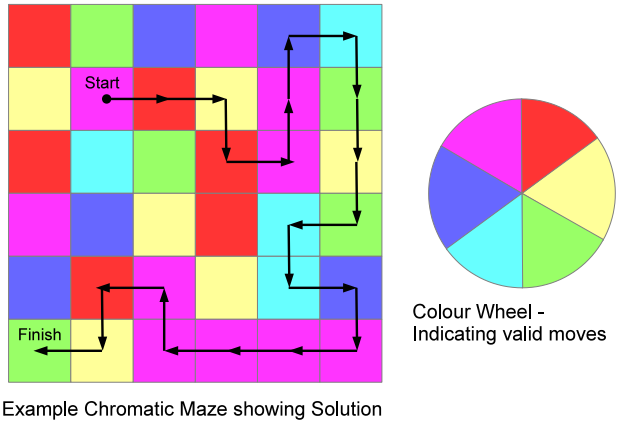

Figure 3. The Chromatic Maze puzzle [13] is used as a benchmark for our research. It has striking similarities to *Roguelike* dungeon maps: they are both based on a 2-D grid of tiles that contain semantic information which can be used to evaluate the usefulness of the level.

A dungeon map for a *Roguelike* game is also a two-dimensional grid of tiles containing semantic information about the level. Instead

of colours the dungeon map contains glyphs that represent the content of the dungeon including walls, floors and monsters. This semantic information can be used to evaluate the level to ensure that the player will be able to navigate the complete dungeon, in much the same way as the Chromatic Maze is evaluated.

3.2.3 *A Naive Approach to Dungeon Building*

A level map for a *Roguelike* game can be represented by a two-dimensional grid, in much the same way as the Chromatic Maze is represented, only it will tend to be much larger. It is not unreasonable to define a *Roguelike* level to comprise a 60-by-60 grid, this is one hundred times the size of the Chromatic Maze. Furthermore, the possible content for each location of the grid will increase from six to something in the order of 20 content artefacts.

The level content can be generated in exactly the same way as for the Chromatic Maze, by randomly placing content at each location in the grid. This method is able to generate every possible level configuration, since it has a direct mapping between the content being generated and the level map. The advantage of this approach is that it has the potential of producing the best level content, however, most of the levels produced will be useless. Evaluating the level content produced to discard the useless levels and find the best levels would be complex and expensive in time. A 60-by-60 grid will take far longer to solve then the two hours to solve the 21-by-21 Chromatic Maze [13].

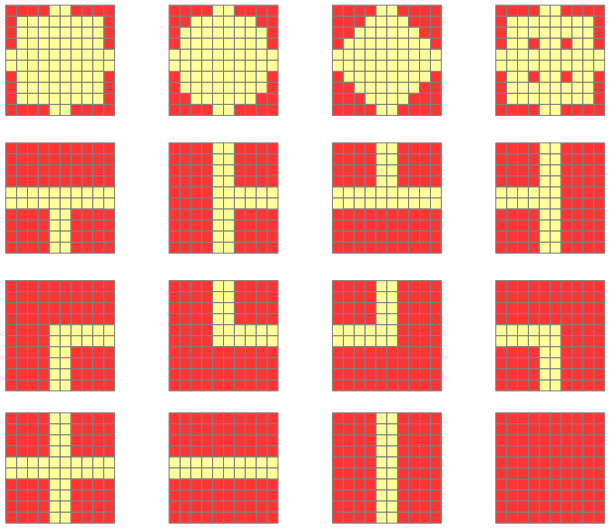

Figure 4. Blocks for Dungeon Building: A set of building blocks are derived that can be placed in the design space to construct a dungeon. They represent four different types of room layout, the four t-junction passages, the four corner passages, the crossing passage, the two straight passages and a blank room/passage. These blocks are of a known size and shape, therefore they can easily be positioned so that they do not overlap.

3.2.4 *Building Blocks for Dungeons*

The naive approach randomly generates content at each grid location, this leads to completely random structures appearing in the level content. An evaluation of such level content, using a step-by-step walk through the level similar to that used in the Chromatic Maze, will not evaluate the quality of the structures produced. This approach would require a much more sophisticated evaluation technique that would be even more costly in terms of the amount of time that would be required to solve than the relatively simple evaluation technique used for the Chromatic Maze.

An alternative approach is to create some structured components to represent rooms and corridor sections, and then randomly position these in the design space. The components could be prefabricated or generated by another PCG technique, to form a set of building component templates of known sizes and shapes that can be utilised to construct a dungeon.

Although placement is random it is relatively easy to prevent overlaying components, and distributing them within the design space. A more difficult issue is ensuring the positioning of the components is such that rooms and corridor sections line up correctly to provide a useful extended dungeon. The evaluation of such a level would be much easier than evaluating a naiver level, however it would still require a much more sophisticated technique than that used for the Chromatic Maze puzzle.

Figure 4 suggest some basic building blocks could be used to generate a level structure. They represent four different types of room layout, the four t-junction passages, the four corner passages, the crossing passage, the two straight passages and a blank room/passage.

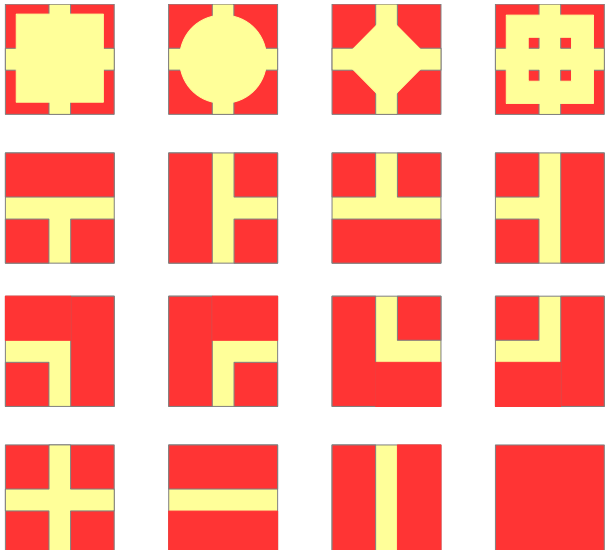

Figure 5. Components for Dungeon Building: The building blocks are represented as logical components in the AnsProlog program. The design space is divided into a number of regions with each region being assigned a logical component.

3.2.5 *A Structured Approach to Dungeon Building*

To alleviate this problem further, a structured approach is adopted to restrict the placement of the dungeon components. The level grid is divided up into a number of equally sized regions, allowing for two phases of PCG. The first phase of PCG decides which components are located in each region, the second level of PCG determines how

that content is positioned within the region. If we take the 60x60 grid example, this can be broken down into a 6x6 grid of regions, where each region is now a 10x10 grid of tiles.

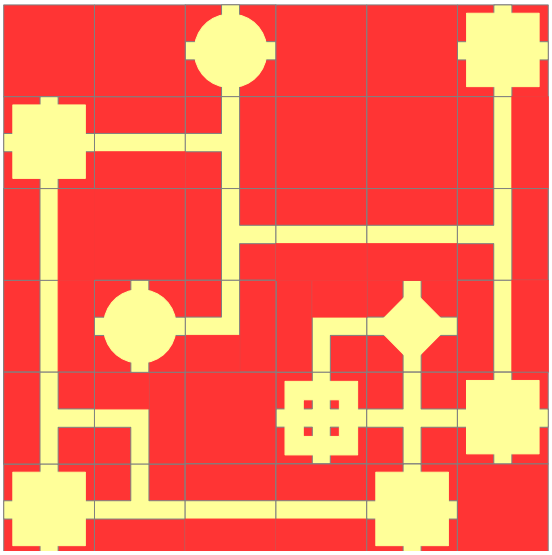

Figure 6. Structured Dungeon Building: A logical representation of the dungeon structure is built by assigning logical components to each of the regions of the design space. The approach allows the AnsProlog program to validate and evaluate the structure of the dungeon layout at a high level.

The first phase of PCG now equates to the same order of magnitude as the Chromatic Maze, depending on the number of different content components available. At this level the PCG process is deciding if the region contains a room or a corridor section, and if so what type of room or corridor section to use. The generation process will also perform some tests to ensure that rooms and corridor sections fit together properly. The room/corridor type will define where the walls and floor are within the region, this provides the overall dungeon environment. An illustration of a dungeon layout for phase one of the PCG is shown in figure 6. This as a visual representation of the logical contents of each of the regions, a rendering of the dungeon where each tile in the 2-D level map is assigned a ASCII glyph is performed during phase two of the PCG.

During the first phase there is a second stage of PCG where the random allocation of artefacts such as monsters and pick-ups is performed for each region (room/passage).

A second phase of PCG will render the detailed content of each region, this is performed in two stages, as with the first phase. The first stage is to generate the environment for each region (room or passage), providing the structure to that part of the dungeon. The second stage places the artefacts, such as monsters and pick-ups, in valid locations within the room/passage. Figure 7 depicts the dungeon level on completion of phase two (note that only stage one of both phases has been performed), with an actual rendering given in figure 2.

3.3 Hierarchical Procedural Content Generation

In the previous section we described the scalability issue as it pertains to large design spaces such as level maps for roguelike games. The method outlined to avoid the problem is to apply a structured

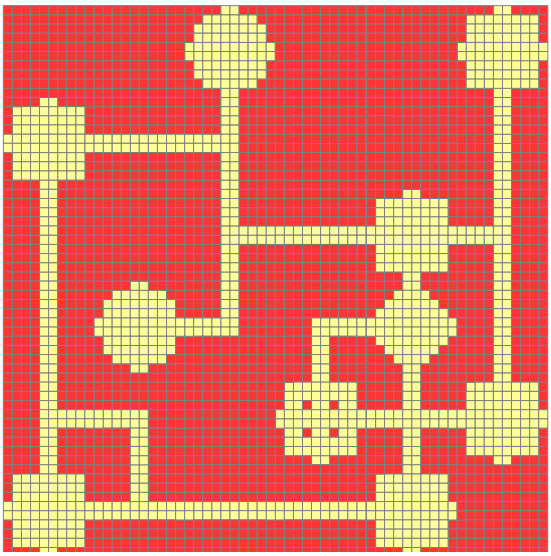

Figure 7. Depiction of Rendered Dungeon Map following Phase 2 of PCG. Using the blocks from figure 4 the 6-by-6 grid of regions from figure 6 can be rendered to achieve a 60-by-60 level map of the complete dungeon. The actual rendering using ASCII character glyphs is given in figure 2

hierarchy to the design space by splitting the design space into a number of equal regions. This results in two distinct phases of PCG, where the first phase determines the properties associated with each region, then the second phase deals with the detail of content placement within the region. This section will outline the method used to implement the first phase of PCG.

	Stage 1 - Environment	Stage 2 - Artefacts
Phase 1	Rooms/Passages for Regions	Content of Rooms/Passages
Phase 2	Room/Passage details	Positioning of Artefacts

Table 1. PCG: Stages and Phases. The PCG process has been split into two phases: phase one generates the logical content of each region of the design space, and phase two performs the detailed placement of content in each region. Both phases have two stages: the first stage generates the level environment content, the second stage generates the level artefacts (e.g. monsters)

3.3.1 *Applying Structure to the Design Space*

The following AnsProlog code segment shows how room are assigned to regions within the design space. The structural hierarchy is defined by splitting the design space into a number of equal regions, these are defined by *raxes* that divides the space into *x_region* columns and *y_region* rows. Three types of room are specified: special, normal and corridor. The generate section uses choice rules to create nine rooms and 18 corridor sections in random regions. The integrity constraints in the test section remove solutions that have regions that contain multiple room sections.

```
% --- Define ------------------------------------------
#const n_regions = x_regions * y_regions.
#const n_rooms = n_regions/4.
#const n_special = 3.
#const n_normal = n_rooms - n_special.
#const n_corrs = n_regions - n_rooms.
```

```
raxes(0..x_regions-1,0..y_regions-1).
types_of_room(special; normal; corridor).
special_type(tomb_of_fear;
             chamber_of_death;
             pit_of_flame).
% --- Generate ----------------------------------------
1{special_rooms(Rx,Ry,S):raxes(Rx,Ry)}1 :- special_type(S).
rooms(Rx,Ry,special) :- special_rooms(Rx,Ry,S).
n_normal{rooms(Rx,Ry,normal) :raxes(Rx,Ry)}n_normal.
n_corrs{rooms(Rx,Ry,corridor):raxes(Rx,Ry)}n_corrs.
% --- Test --------------------------------------------
:- rooms(Rx,Ry,T1), rooms(Rx,Ry,T2), T1!=T2.
:- rooms(Rx,Ry,normal), rooms(Ru,Rv,normal),
   adjacent_region(Rx,Ry,Ru,Rv).
:- rooms(Rx,Ry,normal), rooms(Ru,Rv,special),
   adjacent_region(Rx,Ry,Ru,Rv).
:- rooms(Rx,Ry,special), rooms(Ru,Rv,normal),
   adjacent_region(Rx,Ry,Ru,Rv).
:- shapes(Rx,Ry,S1), shapes(Rx,Ry,S2), S1!=S2.
:- special_rooms(Rx,Ry,S1), special_rooms(Rx,Ry,S2),
   S1!=S2.
```

3.3.2 *Generating Building Components*

Following the allocation of rooms to regions it is necessary to assign those rooms a shape, this will not only affect the way the room component looks, but determines legal movement through the dungeon. There are four room shapes: rectangular, oval, diamond and open, and 12 corridor shapes to represent t-junctions, corners, straight sections and crossing sections. In the generate section each room is randomly assigned a rooms shape, and each corridor section is randomly assigned a corridor shape. A test is made to ensure that no solution has any rooms (regions) that contain more than one shape.

```
% --- Define ------------------------------------------
room_shapes(rectangular; oval; diamond; open).
corridor_shapes(cross; s_hor; s_ver; blank;
                t_tlr; t_ltb; t_blr; t_rtb;
                el_lt; el_rt; el_lb; el_rb).
% --- Generate ----------------------------------------
1{shapes(Rx,Ry,S) : room_shapes(S)}1 :- rooms(Rx,Ry,T),
                                        T!=corridor.
1{shapes(Rx,Ry,S) : corridor_shapes(S)}1 :- rooms(Rx,Ry,T),
                                            T==corridor.
% --- Test --------------------------------------------
:- shapes(Rx,Ry,S1), shapes(Rx,Ry,S2), S1!=S2.
```

3.3.3 *Testing for Unwanted Configurations*

The *rooms* and *shapes* predicates defined above form the concept of a building component. So far we have populated the regions of the design space with these building components, however their place meant has been completely random. Integrity constraints are used to ensure that these building components tie up with their neighbouring components. These tests remove any solutions where the adjacent building components do not have compliant access routes. Note that for sake of space only the integrity constraints for the *cross* building component is listed, but you get the idea.

```
% --- Define ------------------------------
up_sections   (cross; s_ver; t_tlr; t_ltb; t_rtb;
  el_lt; el_rt; rectangular; oval; diamond; open).
down_sections (cross; s_ver; t_blr; t_ltb; t_rtb;
  el_lb; el_rb; rectangular; oval; diamond; open).
left_sections (cross; s_hor; t_tlr; t_ltb; t_blr;
  el_lt; el_lb; rectangular; oval; diamond; open).
right_sections(cross; s_hor; t_tlr; t_blr; t_rtb;
  el_rt; el_rb; rectangular; oval; diamond; open).
% --- Generate ----------------------------
% --- Test --------------------------------
:- shapes(Rx,Ry,cross), shapes(Rx+1,Ry,S),
   not right_sections(S).
:- shapes(Rx,Ry,cross), shapes(Rx-1,Ry,S),
   not left_sections(S).
:- shapes(Rx,Ry,cross), shapes(Rx,Ry-1,S),
   not up_sections(S).
:- shapes(Rx,Ry,cross), shapes(Rx,Ry+1,S),
   not down_sections(S).
...
```

3.3.4 *Evaluating Level Suitablility*

By this stage we should have solutions that contain building components placed in regions such that access is possible between adjacent regions depending on the building component type. The final step to ensure that the level is practical is to ensure that the player is able to navigate around the dungeon from the start location to the end location. Dungeons in *Roguelike* games tend to have a number of level exits that are located throughout the dungeon. However we envisage that a good level will have an objective to achieve, this is represented in its simplest form as a special room that contains a tempting piece of treasure alongside the level boss and his henchmen. Moreover, the placement of stairs between levels will be allocated by the level generator.

```
% --- Define ------------------------------
special_room_types(tomb_of_fear;
                   chamber_of_death;
                   pit_of_flame).
completed_at(S) :- end(Rx,Ry), move_to(Rx,Ry,S).
completed :- completed_at(S).
% --- Generate ----------------------------
1{objective_room(O) : special_room_types(O)}1.
1{special_rooms(Rx,Ry,S) : raxes(Rx,Ry)}1
    :- special_room_types(S).
1{ start(Rx,Ry) : raxes(Rx,Ry) }1.
end(Rx,Ry) :- special_rooms(Rx,Ry,S), objective_room(S).
% --- Test --------------------------------------------
:- not completed.
:- completed_at(S), S<n_regions/2.
% --- Evaluate ----------------------------------------
move_to(Rx,Ry,0) :- start(Rx,Ry).
move_to(Rx,Ry,S) :- step(S),
                    move_to(Px,Py,S-1),
                    adjacent_region(Px,Py,Rx,Ry),
                    move_region(Rx,Ry,Px,Py),
                    0{ move_to(Rx,Ry,0..S-1) }0.
```

4 PRELIMINARY RESULTS

The hierarchical PCG approach, where an initial phase generates the structure of the dungeon followed by a second phase that then filled in the detail, is key to providing a method that can generate content for large design spaces in a timely manner. The emphasis of our current research is on the first phase of procedural content generation. The design space is split into a number of regions with the objective being to assign building components, in the form of rooms and passages, to each region. For example, a 60-by-60 tile design space is divided into 6-by-6 regions each of 10-by-10 tiles. The first phase has now been reduced to solving for a 6-by-6 tile design space, which is similar to the Chromatic Maze puzzle. Running the system on a single core of a 2.66 GHz Intel Core 2 Duo, we obtain the solving times show in table 2 for the first phase of our method.

Regions	1 Answer Set	10 Answer Sets	100 Answer Sets
6x6	6.100	6.110	6.240

Table 2. Results: Time to Generate Answer Sets (in Seconds). The times shown are for the AnsProlog program to generate the answer sets for the first phase of PCG.

A time of several seconds is quite long if the intent is to generate the levels online in real-time. However, there are engineering solutions that can help to reduce the effect of this time period. An obvious solution is to generate the level content whilst the player is still occupied on the previous level or whilst they are distracted with an animated cut scene. Another method is to generate smaller sections of the level, which should be quicker to produce, and piece them together when the player enters that part of the dungeon.

The other approach is to generate the levels offline prior to playing the game. In this way the level data is ready to be loaded at any point during play. However, it would be useful from the level designers point of view to have a system where the level maps can be viewed whilst the level designer is adjusting the level parameters. This interactive process would be an invaluable asset to the level designer, aiding them to produce good quality level content in reasonable time.

5 CONCLUSION AND FUTURE WORK

Answer Set Programming is a powerful tool that can be used to tackle NP-hard problems. However, when dealing with large problem spaces, such as the design space for a roguelike dungeon, a naive approach can be very costly in terms of time. Procedural content generation needs to operate in real-time if the content generated is to be generated and used online. Even where we may have the luxury of generating content offline it is important that the tools used either provide an interactive experience to the level designer, or provide content that has be sufficiently evaluated.

The approach adopted here to avoid the large problem spaces, is to restrict the problem space by applying structure in the form of regions. The overall design space is divided into a number of equal regions, where content is assigned to each region using a random PCG technique. These regions relate to rooms or passages within the dungeon. This hierarchical method effectively reduces the design space significantly reducing generation times form many hours to just a few seconds. The process uses two phases of PCG where the first deals with the random distribution of content between regions (rooms/passages), and the second phase deals with the detailed placement of content within the region, most of which will be fixed.

There are a number of directions that we envisage taking this research. Adopting a two phased approach to PCG is key to addressing the scalability issue, where the first phase has been presented here. The second phase, where the content of each location in the design space is determined, can be achieved using ASP, however there is an opportunity to assess the potential of utilising the strengths of another PCG techniques to produce a PCG hybrid.

The parameter interface with the level designer could be developed to enhance the structure and content of the dungeon maps. The development of a proper user interface or scripting format would go a long way to improve this potential.

Finally, the concept of splitting the design space into regions could be developed to be more flexible so as to produce more intricate dungeon environments. The assessment of the playability of the levels generated would provide useful insight into how the dungeon maps can be improved.

REFERENCES

[1] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang, 'Interactive procedural street modeling', in *ACM Transactions on Graphics (TOG)*, volume 27, p. 103. ACM, (2008).

[2] Jonathon Doran and Ian Parberry, 'Controlled procedural terrain generation using software agents', *Computational Intelligence and AI in Games, IEEE Transactions on*, **2**(2), 111–119, (2010).

[3] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner, 'Answer set programming: A primer', in *Reasoning Web. Semantic Technologies for Information Systems*, eds., Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate Schmidt, volume 5689 of *Lecture Notes in Computer Science*, 40–110, Springer Berlin / Heidelberg, (2009).

[4] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider, 'Potassco: The potsdam answer set solving collection', *AI Communications*, **24**(2), 107–124, (2011).

[5] Lawrence Johnson, Georgios N Yannakakis, and Julian Togelius, 'Cellular automata for real-time generation of infinite cave levels', in *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, p. 10. ACM, (2010).

[6] George Kelly and Hugh McCabe, 'Citygen: An interactive system for procedural city generation', in *Fifth International Conference on Game Design and Technology*, pp. 8–16, (2007).

[7] Mathieu Larive and Veronique Gaildrat, 'Wall grammar for building generation', in *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, GRAPHITE '06, pp. 429–437, New York, NY, USA, (2006). ACM.

[8] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool, 'Procedural modeling of buildings', *ACM Trans. Graph.*, **25**(3), 614–623, (July 2006).

[9] Yoav IH Parish and Pascal Müller, 'Procedural modeling of cities', in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 301–308. ACM, (2001).

[10] Ken Perlin, 'An image synthesizer', *SIGGRAPH Comput. Graph.*, **19**(3), 287–296, (July 1985).

[11] Adam M Smith, Eric Butler, and Zoran Popovic, 'Quantifying over play: Constraining undesirable solutions in puzzle design', (2013).

[12] Adam M Smith and Michael Mateas, 'Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games', in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pp. 273–280. IEEE, (2010).

[13] Adam M Smith and Michael Mateas, 'Answer set programming for procedural content generation: A design space approach', *Computational Intelligence and AI in Games, IEEE Transactions on*, **3**(3), 187–200, (2011).

[14] Julian Togelius, Tróndur Justinussen, and Anders Hartzen, 'Compositional procedural content generation', in *Proceedings of the FDG Workshop on Procedural Content Generation*, (2012).

[15] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky, 'Instant architecture', in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pp. 669–677, New York, NY, USA, (2003). ACM.