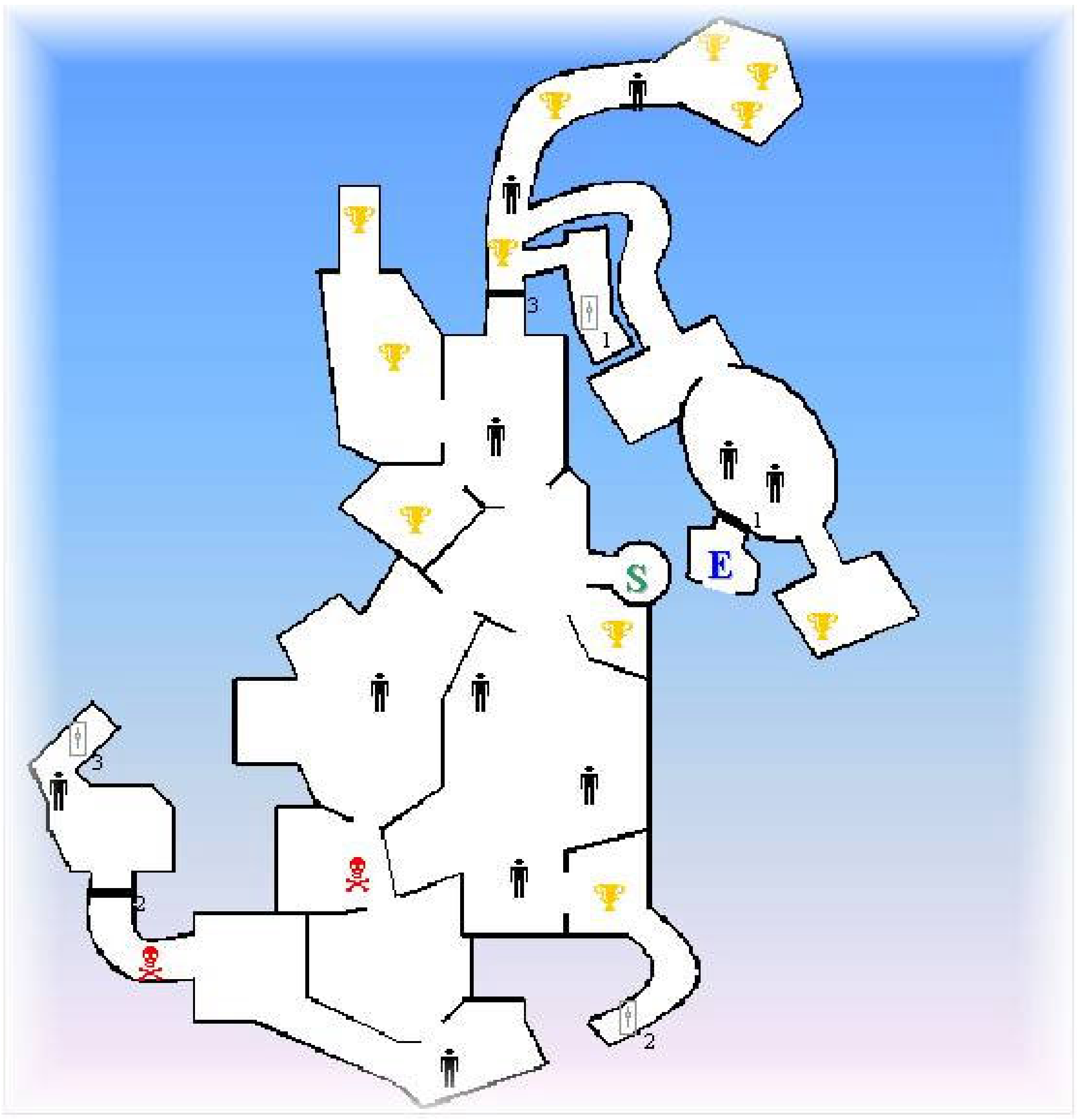
Automatic Generation of Dungeons for Computer Games

Author – David Adams

Date - 28/05/2002

Supervisor – Michael Mendler

This report is submitted in partial fulfilment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by David Adams.



Declaration

All sentences or passages quoted in this dissertation from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this dissertation have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Name:

Signature:

Date:

Abstract

The genre of dungeon games, or first-person shooter games as they are more commonly known, has emerged over the last ten years to become one of the most popular types of computer game. At present, the levels in this type of game are generated manually, which is a very expensive and time-consuming process for games companies.

If levels could be generated automatically then this would not only reduce development costs, but allow levels to be generated at run-time, giving game players a new playing experience each time a game was played and greatly improving the replayability of the game.

In this project, a technique known as graph grammars will be used in order to allow descriptions of randomly generated game levels to be created automatically. As part of the project, algorithms to assess the size, difficulty and fun-value of a level will be developed, to allow individual levels to be tailored to particular requirements. This project is being undertaken in cooperation with Infogrames.

Figure 0a: Screenshot from the dungeon game *Half Life* [57]

Acknowledgments

I would like to thank the following people for their help during this project: -

Ian Badcoe, of Infogrames, for sparing the time to describe the objectives of the project to me.

Marian Gheorghe, my second examiner, for his comments on this report.

Michael Mendler, my project supervisor, for his support throughout the project.

Contents

Chapter 1 – Introduction 1
Chapter 2 – Literature Review 3
2.1 DUNGEON GAMES 3
2.1.1 Description *3*
2.1.2 History *3*
2.2 AUTOMATIC CONTENT GENERATION IN GAMES 5
2.2.1 Content Generation in Dungeon Games *5*
2.2.2 Current Automatic Content Generation in Games *5*
2.2.3 Fractal and Bitmap Terrain Generation *6*
2.2.4 The Context-Free Grammar Approach *7*
2.3 GRAPH GRAMMARS 9
2.3.1 Introduction *9*
2.3.2 Algorithmic Node Replacement Systems *10*
2.3.3 Algebraic Node Replacement Systems *11*
2.3.4 Controlling Rule Application *12*
2.3.5 Summary *13*
2.4 GRAPH GRAMMAR TOOLS 13
2.4.1 Task Specific Tools *13*
2.4.2 General Programming Tools *14*
2.5 LEVEL DESIGN 16
2.5.1 Game Theory *16*
2.5.2 Psychology and Game Design *17*
2.5.3 Good Level Design *18*
2.6 SUMMARY 19
Chapter 3 – A General Graph Grammar System 21
3.1 DESIGN 21
3.1.1 Specific Requirements *21*
3.1.2 Graph Grammar Approach and Language *21*
3.1.3 Class Design *21*
3.1.4 Algorithm Design *23*
3.2 EXAMPLE 24
3.3 THE POWER OF CONTEXT-FREE GRAPH GRAMMARS 25
Chapter 4 – The Dungeon Generation System 27
4.1 A BASIC STRATEGY TO GENERATE LEVELS 27
4.1.1 Search Concepts *27*
4.1.2 Design *28*
4.1.3 Example *31*
4.2 ESTIMATING PARAMETERS 31
4.2.1 Size *33*
4.2.2 Difficulty *35*
4.2.3 Fun-value *35*
4.3 CREATING GOOD LEVELS 36
4.3.1 The Pointless Area Problem *36*
4.3.2 Design *37*
4.3.3 Example *41*
4.4 A GRAPHICAL USER INTERFACE 42
4.4.1 Introduction *42*
4.4.2 Design *42*

4.4.3 Example ... *44*
4.5 SUMMARY ... 44
Chapter 5 – Evaluation ... 46
5.1 RULES FOR AN EXAMPLE GAME ... 46
5.1.1 Introduction ... *46*
5.1.2 Modelling the Game ... *46*
5.1.3 Example ... *49*
5.1.4 Conclusions ... *50*
5.2 EFFICIENCY ... 51
5.2.1 The Problem ... *51*
5.2.2 Improving Rule Application ... *51*
5.2.3 Improving Dealing With Pointless Areas ... *51*
5.2.4 Results ... *52*
5.3 OTHER FACTORS ... 54
Chapter 6 - Conclusion ... 55
6.1 PROJECT SUMMARY ... 55
6.2 SUCCESS OF THE PROJECT ... 55
6.3 FUTURE WORK ... 56
6.4 THE FUTURE OF DYNAMIC CONTENT GENERATION IN GAMES ... 57
References ... 59

Chapter 1 – Introduction

In the rapidly changing world of computer games, game players are becoming ever more demanding and new hardware technology is constantly pushing back the limits of what can be achieved in games. This has led software developers to produce games that are more complicated and with better graphics than ever before.

Due to this greater complexity, increasing time and effort must be put into levels, which are the units that make up a game. It is estimated that it takes a development team a year to produce a game with forty hours worth of content [21]. This is likely to grow as more features are added to games and as the graphics improve.

It would therefore be desirable if a system could be developed that can take as input a set of rules defining properties of the levels that must be produced for a specific game, and automatically create levels for that game. This system could work in one of two ways. The simplest of these is to generate levels during the development of a game. Here, the design team would select the best levels, improve them if they wished, and then put them into the game.

Using the above type of generation, the game player would be unaware that the levels of the game had in fact been generated automatically. The other option is for levels to be dynamically created when the player runs the game. This would not only save the software developer time and money, but would also make people more likely to play the game again after they had completed it, because the next time it was played the levels would be different, adding immense variety to the game.

Figure 1a: Randomly generated terrain in *SimCity 3000* [31] makes the game highly replayable.

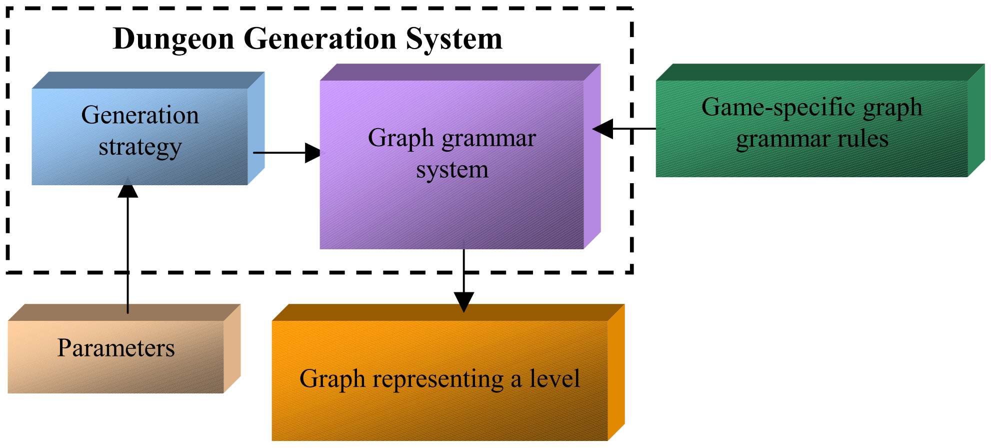
The development of a system to automatically generate levels is the aim of this project. At this early stage, the system will be designed primarily to generate levels during game design rather than dynamically. Another constraint on the project is that the focus will be on generating levels for a particular genre of computer game, known here as dungeon games.

The technique that will be used to generate the levels is a type of grammar known as a graph grammar. Put simply, graph grammars concern the generation of graphs from rules, rather than the generation of strings from rules, as is the case in most grammars, such as context-free grammars.

A diagram showing the main features of the system that will be developed is shown in Figure 1b. The program will use a general graph grammar system that allows graphs to be manipulated. The user of the system will input a set of rules to randomly generate a level for a particular game.

A set of parameters will also be supplied to the system. These will specify the size and difficulty that is required for the generated level. A generation strategy should then use these parameters and the rules to create a level, which is output as a graph.

It is not intended for the graph output by the system to specify a level in so much detail that it could instantly be attached to the game it was designed for and played. As will be explained in Chapter 2, there are two ways a level can be defined. A *geometric* description is a physical description of the level, whereas the Dungeon Generation System will output a *topological* description of a level, which is a level definition saying nothing about physical sizes that is just concerned with the order in which objects in the level are encountered.



Dungeon Generation System
Generation strategy
Graph grammar system
Game-specific graph grammar rules
Parameters
Graph representing a level

Figure 1b: The Dungeon Generation System

The main aims of the project are as follows: -

1. Find, or if necessary, create a general system to support the graph grammar approach.
2. Create strategies that generate topological descriptions of levels. These levels must:
    - Be random and vary greatly from other levels.
    - Have the difficulty and size that were input as parameters.
3. Develop algorithms to assess the size, difficulty and fun-value of a level.
4. Develop a rule set that allows interesting, fun to play and differing levels to be generated by the Dungeon Generation System.
5. Investigate the power of context-free graph grammars with regard to automatic level generation.

If time allows, the following will also be done: -

1. Develop a system to convert the topological level descriptions into geometric level descriptions.
2. Create complete and playable levels for an actual dungeon game using levels output by the Dungeon Generation System.

An overview of this report will now be given. Chapter 2 contains a literature review that describes in detail the main areas of work related to the project, including a description of dungeon games, graph grammars, current automatic content generation in games and principles of good level design. In Chapter 3 the underlying graph grammar system used in the project is described, and the power of context-free graph grammars is investigated.

Chapter 4 describes the development of the Dungeon Generation System, which is a system that creates topological descriptions of levels through the manipulation of graphs. A basic strategy is created in order to allow levels of a certain size and difficulty to be generated, and specific ways of estimating these parameters are implemented. A more sophisticated strategy for generating levels is then developed that produces more interesting levels, and a graphical user interface is produced to help visualise the levels and make the system more easy to use.

In Chapter 5 the system is evaluated on a number of criteria, and to help assess several of these an example rule set for a real dungeon game is created. The report is concluded in Chapter 6, which discusses the overall success of the project and suggests possible future work.

Chapter 2 – Literature Review

2.1 Dungeon Games

2.1.1 Description

There are many computer game genres, from adventure games to strategy games and simulation games, but this dissertation is concerned with what will be called here dungeon games and are more commonly referred to as first person shooter games.

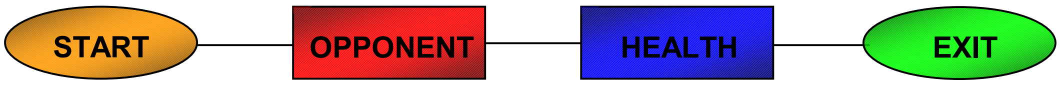
Unlike in most other genres, which use the third person perspective, the character controlled by the player cannot be seen on the screen. This first person perspective allows the player to see the game through the eyes of that character, giving a sense of actually being there.

This immersion in the game is one of the things that make the dungeon game genre so popular, as well as its fast-paced, real-time action, simplicity and excellent graphics with a three-dimensional view. Games of this type tend to have a limited storyline and the character in them usually has to explore an area (often resembling a maze or dungeon) while killing any monsters encountered.

The player usually has available several types of weapons, limited ammunition, perhaps some shielding or armour protecting them from monsters and other hazards, as well as health, displayed as a percentage, which can be decreased from contact with monsters and increased by certain rewards such as medical packs or potions. In order to travel through the game world, keys may have to be found to open doors, switches may have to be activated and puzzles may have to be solved.

As with most computer games, dungeon games are organised into levels. A level can be thought of as a self-contained unit. The player starts at one position in the level and usually has to progress through the physical area that is the current level to reach a certain point marking the end of the level. At this point the level has been completed and the following one can be started. In some games, to complete a level an objective, such as finding an item, marks the end of a level, as opposed to merely reaching a particular point. However, this type of level will not be discussed in this project.

Two terms that will be needed later on are *topology* and *geometry*. The topology of a level describes the order in which things are encountered in travelling through the level. It says nothing about directions or physical sizes. For example, the topological description:



says after starting the level the player encounters a monster, then a health bonus before completing the level.

The geometry of a level is a physical description of the level. This could be in several different levels of detail from a 2D plan showing the rooms, corridors and objects, a 2D plan using exact coordinates, through to a 3D map complete with textures and lighting.

2.1.2 History

Today, dungeon games are classed as a subsection of action games, but they actually have their roots in role-playing games (RPGs). The following description of RPGs contains information from [16]. RPGs are usually set in fantasy worlds. Battles are fought alongside other players, in order to kill monsters to get rewards such as gold. These rewards are then sold for weapons that are more powerful, potions or spells, which are then used to fight even more monsters.

The first RPG was a book called *Dungeons & Dragons*, released in 1974. Many other 'pen and paper' RPGs followed, and in 1979 a company called Sir-Tech developed the first computer game of this type called *Wizardry: Proving Grounds of the Mad Overlord*. This did have first-person perspective but it was not in 3D, the player could not turn and move freely through the environment, and the game was turn-based and not played in real-time.

Over time, the games became more sophisticated and by 1992 games such as *Ultima-Underworld* from Bluepoint/Looking Glass, were now in 3D with real-time movement through any direction, as opposed to just allowing turns of ninety degrees.

From games like these the new genre of dungeon games was born. They were much simpler to play than their RPG counterparts with more action, and they omitted the puzzles that many games players found unexciting.

The first dungeon game, or at least the first one to gain widespread appeal, was *Wolfenstein 3D*, released by id Software as shareware in 1992 [18]. The game's thin storyline involved a prisoner trying to escape from an enemy fortress during World War II.

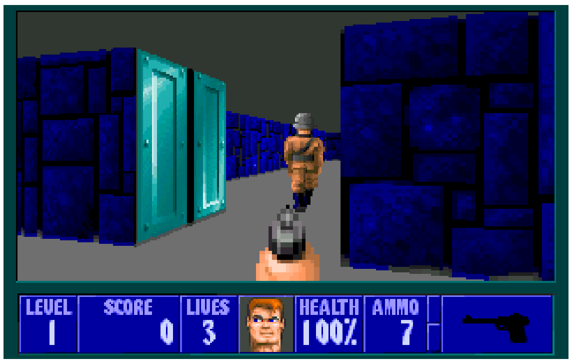


Figure 2.1.2a: Screenshot from *Wolfenstein 3D* [18]

By today's standards, the graphics in Wolfenstein are poor and its levels are rather unimaginative, but at the time it was revolutionary. It won awards for being the best arcade/action game of that year and at the time magazines such as Computer Gaming World called it '*The first game technologically capable of ... immersing the player in a threatening environment.*' [58]

Since then, the genre has expanded greatly to become one of the biggest in the computer games industry. The section below charts the progress of the genre through some of its most significant games, including information from [58]: -

Doom – Id Software (1993) [19]
Doom is one of the most famous games in this genre. Made by the creators of Wolfenstein, it concerns a space marine escaping from a complex full of mutants. With better graphics, bigger weapons, scarier monsters and great level design, it was an instant success.

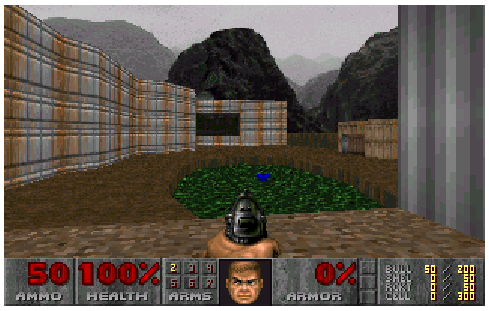


Figure 2.1.2b: Screenshot from *Doom* [19]

Quake – Id Software (1996) [20]
Quake had a similar style to Doom, except it was more advanced. It was the first dungeon game to be in true 3D (in previous games this had not been the case with monsters and other objects).

Jedi Knight – LucasArts (1997) [26]
Set in the Star Wars universe, this game was quite ground-breaking. It had a much more detailed storyline than previous games in the genre, and the story was also non-linear, branching off depending on which path the player chose to take and which abilities he/she gained.

Unreal/Unreal Tournament – Infogrames (1998-1999) [22]
This game was different from the rest in that it was set in the future at a tournament where it did not matter if your character was killed as long as you killed more monsters than anyone else. It had some of the most detailed graphics of any game.

Half Life – Valve (1998) [57]
Half Life is seen by many (including [58]) as being the best dungeon game ever made. It had an interesting and complex storyline involving stopping aliens using a portal to get down to Earth. Its most notable features were the excellent game atmosphere and the improved artificial intelligence of monsters in the game. These monsters could use complex tactics against the player, such as setting traps for them.

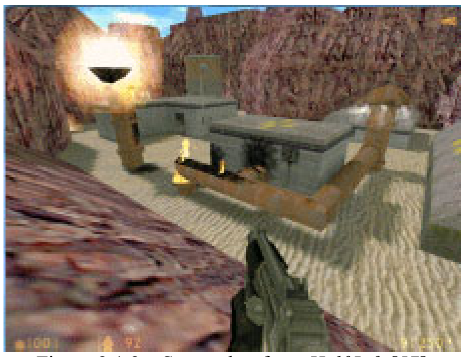


Figure 2.1.2c: Screenshot from *Half Life* [57]

2.2 Automatic Content Generation in Games

2.2.1 Content Generation in Dungeon Games

In current dungeon games, the layout of the levels and the positions of objects within those levels are all done manually. In the past whole games were done by programmers, but in recent times the new job title of level designer [54] has emerged, and it is he or she who is ultimately responsible for level design, although many other people, from programmers working on the artificial intelligence to texture artists, have an input into the process.

A simplified view of the usual procedure (as described in [54]) to create a level is as follows: firstly, the level designer considers what type of level it is, such as a single-player or multi-player level, and then they sketch out a diagram of the level, often called the *map*. An architectural style is decided on, and the designer starts by placing large blocks onto the map to define features such as corridors, landings and other areas. Extra detail is then included, such as to define the exact shape of a room or dimensions of a staircase. Textures are added to complete the geometry. Only at this stage are monsters and other items added.

If the process of level design is to be automated then all of the above stages will have to be done randomly by a computer. The order in which the parts making up a level are currently done should be considered when automating the process, as the strategy used presently is clearly successful and so that a partial automation of the process could be used along with stages done manually.

2.2.2 Current Automatic Content Generation in Games

Although in the dungeon genre of games no levels are generated randomly, this is not the case for several other genres, for example strategy games. An example of such a game is MicroProse's *Civilization II* [36] in which the player starts with a single settler and has to build an empire over time. The Earth-like world in which each game takes place is randomly generated to give a new experience each time the game is played.

The game world is split up into square tiles each of which is of a certain type of terrain, such as grassland, forest or ocean. The terrain type of each square is randomly generated, but done in a realistic way with tiles of the same type tending to congregate, and vast continents instead of intermittent water. Object placement, such as the start position of all the players, is also random.

As stated in [11], without the random content generation, Civilization II, and games like it, would have greatly reduced replayability and people would quickly lose interest in them. Exploring the game world would become uninteresting, as players would already know what they would find, and over time it would become clear what strategies to use, so that the game would have the same outcome again and again.

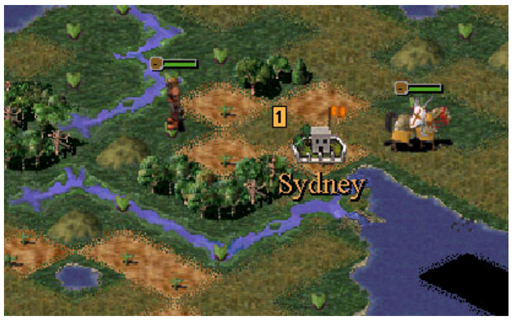


Figure 2.2.2a: Screenshot from *Civilization II* [36]

While many role-playing games have levels that are predefined missions, some have random levels. The type of content generated here is closer to what is required to generate random levels in this project. Games such as *Diablo* from Blizzard [7] have randomly generated dungeons that the game character must journey through on a quest.

However, the generated dungeons were found by many (including [10]) to be too similar. In addition, the generation of these dungeons was only made possible by applying certain constraints such as that all dungeons must be of a certain size and shape and that they cannot contain any puzzles or situations where the player must find a key to open a door.

In the next two sections, two techniques for randomly generating content in games are discussed.

Figure 2.2.2b: Screenshot of a randomly generated dungeon from *Diablo II* [8]

2.2.3 Fractal and Bitmap Terrain Generation

In this section, two similar techniques for generating terrain randomly are discussed. Terrain generation is especially useful in flight simulator games where the player might have to fly over a large area. Automatic terrain generation can save the game developers a lot of time by reducing the amount of height data they have to generate themselves.

Fractal terrain generation is the first technique that will be examined. A more detailed description of this technique can be found in literature such as [23], [30] and [38]. Fractals can be used because terrain is *self-similar*. This means 'magnified subsets of the object look like (or are identical to) the whole and to each other' (from [65]). For example, a mountain on the horizon is not smooth but rough and uneven. If we zoomed in on part of the mountain's slope, it would also look uneven, as would the surface of an individual rock or stone that was part of it.

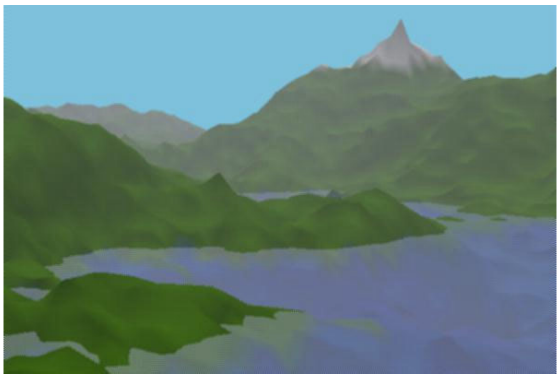


Figure 2.2.3a: Screenshot of randomly generated terrain from *Birdland* [23]. Reproduced with kind permission of Samu Karanko, Helsinki University of Technology

This self-similarity allows fractals to be recursively used to define greater and greater detail on a generated terrain until it is sufficiently detailed. The technique used for this is the edge subdivision algorithm, which is described below (and in [23], [30]). It is a fractal technique because it is applied recursively to repeatedly increase the level of detail, until finally the terrain is sufficiently detailed. Firstly, two random numbers are generated and a line is drawn between them. Then, the midpoint of the line is found, a new node is created at this point, and the node is shifted up or down by a random amount that is proportional to the height difference between the original start and end points of the edge. The new node forms two new lines, and the process is repeated with the midpoints of these lines. When there are enough nodes that the gap between each has shrunk to a certain size, the process stops. The average random displacement should decrease at each level of recursion so the terrain does not appear too jagged.

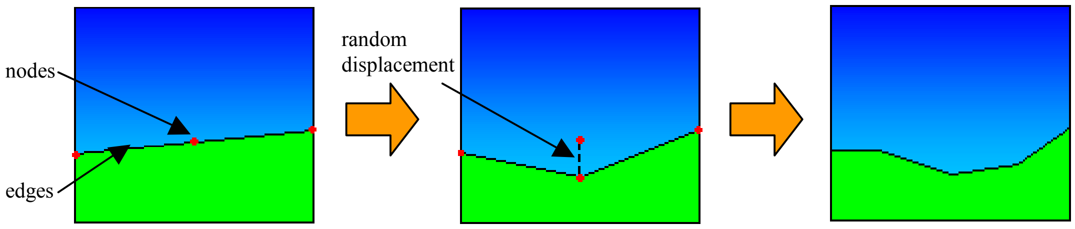


Figure 2.2.3b: Using the edge subdivision algorithm to generate random terrain

Here two-dimensional terrain has been generated, but techniques exist to extend this to three dimensions. One of the best is the Diamond-Square Algorithm, which was originally described in [15]. When all of the height data has been generated, it can be rendered using a particular technique.

Another technique, described in [29], involves using bitmaps rather than fractals to generate the terrain. Unlike fractal terrain generation, this is not a completely random technique; there is already an initial map with general features on it, and the process uses this to generate detailed data for each of the small units that make up the map.

This is often the case with extremely large multi-player maps in that the designers may want to have certain features, but do not care about the precise height of every single square. Indeed, it would be very time consuming to generate data for each square manually, so the technique helps to save a lot of time. The data generated by this could then be enhanced where necessary to make the terrain look better.

To start with, a small bitmap with basic features on it is constructed (Figure 2.2.3c). Then it is scaled so that it is the size of the map required (one pixel representing each tile or unit in the map) (Figure 2.2.3d). Image processing techniques such as convolution, which uses the value of neighbouring pixels to calculate the new value of a certain pixel, are used to smooth the bitmap, and random factors can be used at this stage as well (Figure 2.2.3e).

The bitmap is then converted into the height for each tile on the map by taking the value at the corresponding pixel in the bitmap (assuming the bitmap is a greyscale) and multiplying it by a scalar.

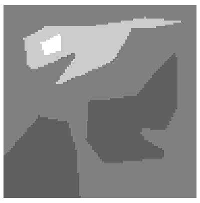


Figure 2.2.3c

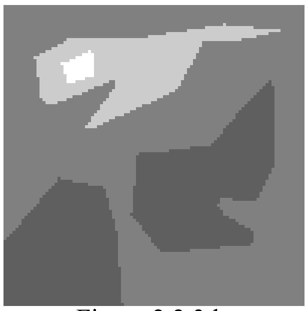


Figure 2.2.3d

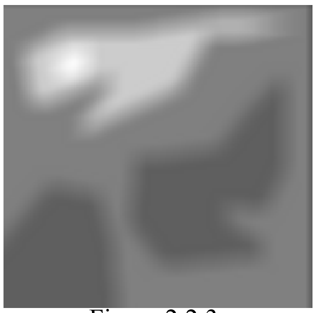


Figure 2.2.3e

The bitmap can be used to generate other data apart from the height of each tile. Each pixel value could correspond to a particular terrain type, such as desert or jungle. The author of [29] even says the technique could be used outside of terrain generation, such as in the placement of objects (such as monsters or doors) on a map. However, it is not clear how this could be achieved, and the author provides no justification for this statement.

2.2.4 The Context-Free Grammar Approach

Another technique is the work done on automatic content generation by Simon Ince in his dissertation ‘Automatic Dynamic Content Generation for Computer Games’ [21]. In this, the technique used to randomly generate the topology of dungeon games employed context-free grammars.

Grammars use rewrite rules to generate all of the strings in a language. For Ince’s project, the language the grammar had to generate was therefore the set of correct (and hopefully interesting) dungeons represented as strings. A (phrase-structure) grammar can be defined formally (from [35]) as an ordered 4-tuple $\langle V, \Sigma, S, P \rangle$ where: -

- V is a finite alphabet, called *variables* or *nonterminal symbols*.
- Σ is a finite alphabet, called *terminal symbols*.
- $V \cap \Sigma = \emptyset$.
- $S \in V$ is the *start symbol*.
- P is a set of ordered pairs $\langle \alpha, \beta \rangle$ called *production rules* such that $\alpha, \beta \in (V \cup \Sigma)^*$ and α contains at least one symbol from V.

The language generated by a grammar is the set of all strings containing only terminal symbols that can be generated by a finite number of applications of production rules.

Context-free grammars are a subclass of phrase-structure grammars. A grammar is context-free if the left-hand side of every production is just a single nonterminal symbol. They are called context-free because their rewrite rules can be applied regardless of the context in which they occur, unlike $xAy \to P$ in which the symbol A can only be rewritten to P if it occurs between an x and a y.

Using the above, a context-free grammar was constructed in the project in which level objects such as ‘opponent’, ‘health’, ‘key’ and ‘door’ were defined as terminal symbols, and nonterminal symbols were introduced in order to define larger subsections of the levels that were rewritten to the terminal symbols by later derivations. These included ‘HEALTH_GROUP’, which was used to specify a group of health objects encountered at the same time, and ‘OBSTACLE’, which could be a group of opponents, or a challenge such as a switch and door, or both.

The levels generated were based on two levels from the game *Doom II*. All levels started with the start symbol ‘LEVEL’ being rewritten to either of these two level structures: -

$$LEVEL \to start\ LEVEL1\ end$$
$$LEVEL \to start\ LEVEL2\ end$$

These were then rewritten themselves in further productions. Simon Ince admits that basing any levels generated on just two level structures was one of the weaknesses of the system, as it led to many of the levels being similar.

The grammar was defined in a file that was parsed by the system. One good feature of the system was that it allowed probabilities to be associated with productions that had the same left-hand side, so that some nonterminal symbols were more likely to be rewritten to a certain string than others. An example of this is the ‘OPPONENT_GROUP’ symbol which is more likely to be re-written as two opponents than one or three: -

OPPONENT_GROUP –30-> opponent
OPPONENT_GROUP –50-> opponent opponent
OPPONENT_GROUP –20-> opponent opponent opponent

The probability of that production being used, out of 100

A difficulty rating was also associated with each terminal symbol to indicate how the presence of that symbol would affect the difficulty of the level. For example, opponents had a positive rating and health had a negative rating (as it made the level easier).

To show how the system represents levels, an example level that could be output by the system is given below: -

```
start opponent opponent JUNCTION (JUNCTION (switch door health bonus)
opponent opponent opponent health bonus bonus) health health bonus
end
```

As can be seen, levels are represented in a linear fashion, and can be thought of as representing regular expressions. The `JUNCTION` keyword is used to indicate a split in the level topology, where the player has a choice of paths to take. The level shown above could be represented geometrically as shown in Figure 2.2.4a.

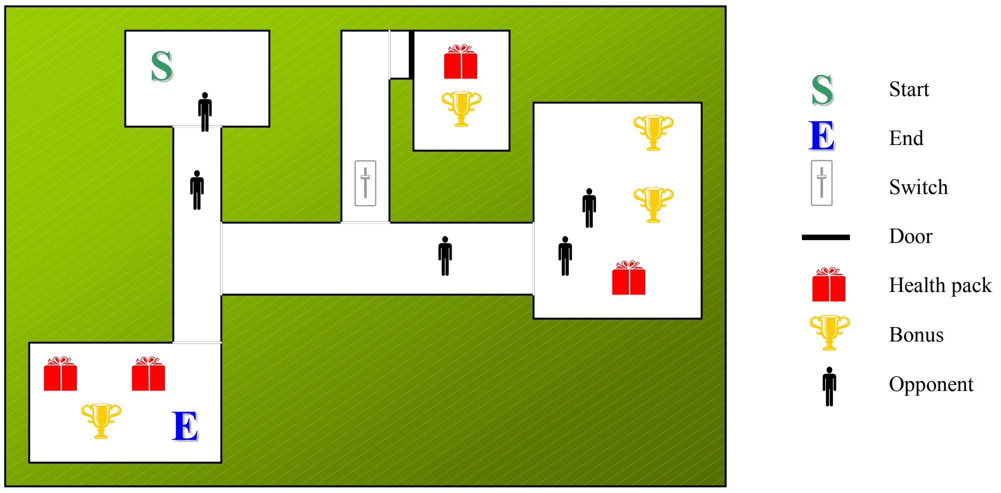


Figure 2.2.4a: A possible geometric description of the example level

A level could be generated either completely randomly, or based on a target difficulty or size. All three methods repeatedly found a nonterminal symbol in the string that represented the level, applied a production rule, and repeated this until only nonterminal symbols were present in the string.

The generation algorithms only differed in deciding which rules to apply. This means deciding which possible derivation of a nonterminal symbol to use, when there is more than one way it can be rewritten. For purely random generation, the production rule was chosen randomly, but took into account the probabilities that were assigned to the production rules.

For a generation based on a target difficulty or size, the system tried to output a level of a size or difficulty input by the user. It did this by calculating the possible final expansion size/difficulty of a nonterminal symbol for all ways that the nonterminal symbol could be rewritten (for example, there are three ways that ‘OPPONENT_GROUP’ can be rewritten), and it then uses these estimates to choose a rule that it thinks will best match the size/difficulty constraint.

The report does not say how the system actually performs the process of choosing a rule based on the sizes or difficulties returned. Trouble was also encountered in estimating the size/difficulty accurately yet still making the result sufficiently random so that different levels would be produced. For instance, if A can be re-written to 3 different strings each containing either a B, C or a D that can themselves be re-written to 3 different strings before containing no nonterminal symbols, then the resulting string could take nine different forms. The estimation algorithm does not know which one of these nine possible final strings will actually be derived, so it cannot accurately estimate what the change in the size or difficulty of the level will be if the rule is applied.

In addition, the measure of a rule's impact on the 'size' of a level was just the estimated number of terminal symbols that the rule would add to the level. Similarly, difficulty was just measured by summing the difficulty ratings assigned to the terminal symbols in the rule. In reality, the size and difficulty of a level are more complex than this, and depend on the content of the rest of the level, not just the local part that is being re-written. For example, an opponent placed in an area of the level that does not need visiting in order to complete the level will increase the difficulty of the level far less than one placed on the main route through the level, because the player can just avoid the former of these two opponents.

The project was successfully finished and a fully functional system was produced. However, there were a number of drawbacks to the system: -

- Levels being produced that are not close to the difficulty or size rating required.
- Levels being produced that are much easier or harder than the difficulty rating estimates (such as those having huge pointless sections that contain lots of opponents but can be avoided by the player).
- Random levels being too similar to each other.
- The grammar is not able to differentiate between multiple key-pair doors (i.e. which key matches which door?). This is because all keys and doors are just assigned the same terminal symbol ('key' and 'door' respectively) and therefore they are indistinguishable from each other because grammar symbols have no unique identity (unlike objects).
- The grammar is not able to cope with complex corridor designs. This is because the JUNCTION keyword is the only means by which a split in the topology can be indicated, and this keyword can only be used to represent two possible paths. In dungeon levels it is not uncommon for there to be four, five or even six branches from a particular corridor or room.

These drawbacks will be taken into consideration during this project, so that the system designed can try to overcome them.

2.3 Graph Grammars

It is clear from the previous section describing context-free grammars that there are major limitations in the use of context-free grammars to generate levels. In the work by Ince (see Section 2.2.4), levels were represented by regular expressions, which do not have the power to adequately represent the complex pattern of connectivity found in levels. In addition, it was seen that the linear way in which context-free grammars rewrite strings means that it is almost impossible to produce a system based on this type of grammar that produces sufficiently random levels while at the same time outputs levels with the size and difficulty required.

The conclusion that must be drawn is that another approach to generating levels must be found, and in this project the method that will be used is one known as graph grammars. Graph grammars are a type of grammar that manipulates graphs instead of strings, and graphs provide a natural way of modelling the complex webs found in levels. Furthermore, graph transformation is much more flexible and controllable than string rewriting, allowing both a greater randomness to be introduced into the procedure and enabling the end result to more closely match those properties (such as size) specified.

2.3.1 Introduction

Phrase structure grammars, such as the previously discussed context-free grammars, have rules to manipulate strings, but graph grammars manipulate graphs instead. The approach was started in the late 1960's by two papers concerning 'web grammars' and 'Chomsky systems for partial orders' ([42] and

[55]). The original uses of graph grammars were in areas such as specifying data types and pattern recognition, but today they have also been applied to fields such as the creation of visual languages and software specification. A general introduction to graph grammars can be found from several sources, including [4], [44], [45] and [50].

The basis of graph grammars is the notion of a graph. A graph is a collection of nodes (or vertices) connected together by edges (or arcs). There are many different types of graphs. Edges can either be directed or undirected. In some graphs labels can be associated with nodes, and some graphs are attributed, where nodes can have variables associated with them. However, a typical formal definition of a graph (from [44]) is as follows: -

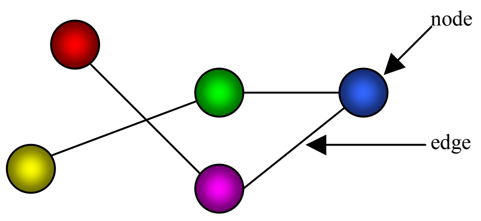


Figure 2.3.1a: An example graph

A labelled and directed graph $G := (V, E, l_V, l_E, s, t)$ is a graph over label sets L_V, L_E with:

$V(G) := V$ is a finite set of vertices,
$E(G) := E$ is a finite set of edges.
$l_V(G) : V \to L_V$ is the labelling function for vertices.
$l_E(G) : E \to L_E$ is the labelling function for edges.
$s(G) : E \to V$ assigns each edge to its source, and
$t(G) : E \to V$ assigns each edge to its target.

A graph grammar (from [44]) is a tuple (A, P) with A a nonempty initial graph and P a set of graph grammar productions. The definition of a graph grammar production varies depending on the formalism being used. However, most approaches have a left-hand side and a right-hand side. In context-free graph grammars, the left-hand side of all productions must be a single node, whereas in context-sensitive graph grammars, both the left and right –hand sides of productions are graphs.

Graph transformation is the process of repeatedly applying productions to a graph in order to generate a new graph. A production can be applied if a match of the left-hand side can be found in the host graph. This is called a *redex*. The nodes in the match are replaced by those on the right-hand side of the production, and the new nodes are connected via edges to the rest of the host graph.

Just as there are many different types of graphs, there also exist many different graph grammar formalisms, which vary in the constraints (if any) that they place on productions, the way rules are applied and how the process of rule application is controlled.

All types of graph grammars fall into one of two categories based on how productions are applied. In *algebraic* (gluing) approaches, *context elements* are used to attach new nodes to the rest of the graph and the basis of these is in category theory, which is a mathematical theory of structures that describes how structures relate to one another [27]. *Algorithmic* (connecting) approaches are based on set theory and use embedding rules instead of context elements. These approaches are discussed in the following subsections.

2.3.2 Algorithmic Node Replacement Systems

In the algorithmic, or connecting, approach, when a match in the host graph is found, all edges connecting the nodes in the match to the rest of the host graph are deleted, and new edges are used to connect the nodes added to the host graph to the rest of it [50]. How the new nodes are connected to the rest of the graph is called the *embedding mechanism*, and this is specified by *connection instructions*.

One simple algorithmic graph grammar system is the NLC (Node Label Controlled) mechanism [50]. In this, the connection instructions are all specified via node labels and the left-hand side of productions must be a single node called the *mother node*.

How productions are applied is best illustrated with an example. Figure 2.3.2a shows one production that turns a node with label ‘S’ into two nodes with labels ‘a’ and ‘b’ that are connected via an edge.

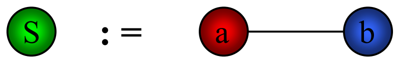


Figure 2.3.2a: An example graph grammar production

The host graph to which the production will be applied is shown in Figure 2.3.2b. There is only one node in this graph with the label ‘S’, so there is only one way in which the above production could be applied. The ‘S’ in the graph is the mother node mentioned earlier. A term that will be needed later is the *neighbours* of the mother node. These are simply the nodes that are connected to the mother node by an edge, which are the nodes labelled ‘b’, ‘c’ and ‘d’ in Figure 2.3.2b.

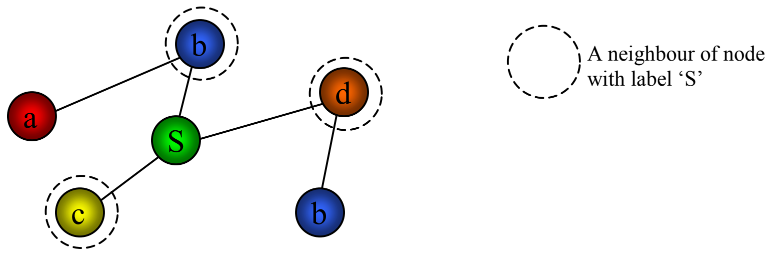


Figure 2.3.2b: The host graph

Firstly, ‘S’ is removed, since it does not appear in the right-hand side of the production. When ‘S’ is removed, so are any edges connecting it to other nodes. Then nodes ‘a’ and ‘b’ are added. The graph would now look like this: -

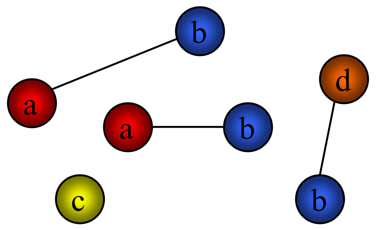


Figure 2.3.2c: The host graph after node replacement

Now all that needs to be done is to use an embedding mechanism to connect the new nodes to the rest of the graph. In the NLC approach, each connection instruction is an ordered pair, and all the connection instructions form a *connection relation*. In this example the connection relation will be {(c,a)(b,b)}. This means that:

- For each node with label ‘c’ in the host graph that was a neighbour of the mother node, it is connected via a new edge with every new node with label ‘a’.
- For each node with label ‘b’ in the host graph that was a neighbour of the mother node, it is connected via a new edge with every new node with label ‘b’.

Thus, the graph after the production has been fully applied would look as in Figure 2.3.2d: -

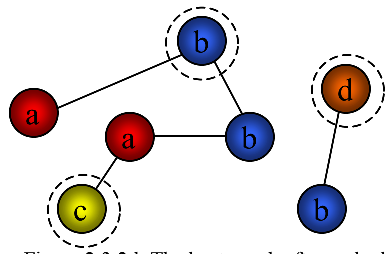


Figure 2.3.2d: The host graph after embedding

A disadvantage of this approach is that all productions use the same set of connection instructions, which means that the nodes generated by all productions are connected to the rest of the graph in the same way. This is not the case for algebraic node replacement systems, which are discussed next.

2.3.3 Algebraic Node Replacement Systems

In the algebraic, or gluing, approaches, node replacement works in a similar fashion to that shown above for the algorithmic approaches [50]. Any matching nodes in the host graph that are not also present in the right-hand side of the production are deleted and any nodes that are on the right-hand side but not on the left-hand side get added to the host graph.

In a production, there may be several nodes all with the same label. The graph may even not have labels. How does the system know which nodes in the left-hand side are the same nodes in the right-hand side, and which ones are just to be added? This problem is solved by a *morphism*, which can be thought of simply as a relation between nodes on the left-hand side of the production and nodes on the right-hand side of the production.

On graph diagrams, this can be illustrated by giving nodes numbers and with matching nodes having the same number. The labels have been omitted to make the diagram look less cluttered, but nodes with the same label are the same colour and shape. Therefore, for the production in Figure 2.3.3a, Node 2 is on both sides of the production so it will neither be deleted nor added to the host graph. Node 1 appears on the left-hand side but not on the right-hand side, so it will be deleted from the host graph, and Node 3 appears on the right-hand side but not on the left-hand side, so it will be added to the host graph.

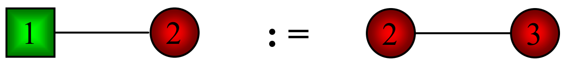


Figure 2.3.3a: A production showing matching nodes

Although node replacement works in a similar way in both of the two approaches, they differ in how the new nodes are embedded into the rest of the graph. A technique called *pushout* is used to glue the new nodes to the rest of the graph. Instead of having an embedding mechanism that uses new edges, *context elements* are used to bridge the gap between the new nodes and the rest of the graph.

A context element is a node that appears on both the left and right –hand sides of a production, such as Node 2 in the example above. Having such nodes means that new nodes are automatically connected to the rest of the host graph. For example, the match for Node 2 is already in the host graph. A Node 3 is created and it is attached to the match for Node 2 so Node 3 is already attached to the rest of the graph, using the node matching Node 2 as a link.

There are two main approaches in algebraic node replacement. These are the double-pushout (DPO) approach and the single-pushout (SPO) approach [50]. The main difference between these can be illustrated by applying the production above to the following graph: -

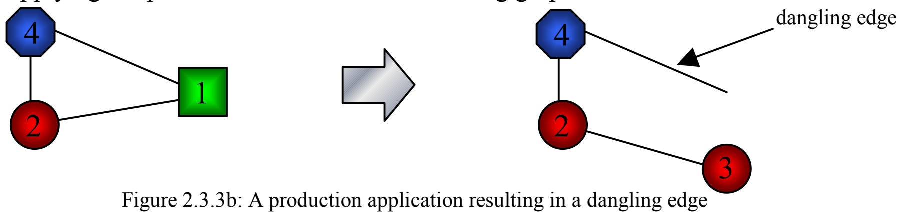


Figure 2.3.3b: A production application resulting in a dangling edge

When the node in the host graph that matches Node 1 is deleted there remains what is called a *dangling edge*. This is the edge that used to go between Node 4 and the node matching Node 1. However, it no longer goes between two nodes. In the DPO approach, if a rule specifies the deletion of a node n, it must also state that all edges connected to n must also be deleted. This was not specified in the example above, so in the DPO approach the rewrite above would not be allowed to occur. In the SPO approach, the rewrite would be allowed, as all dangling edges are automatically deleted.

Both of these approaches to dangling edges have an advantage. By making the user explicitly specify which edges are to be deleted, the DPO approach ensures that edges cannot be accidentally deleted and that nodes cannot accidentally be left unconnected from the rest of the graph. Alternatively, the SPO approach has the advantage of making productions simpler to write because they do not have to include edge deletion information.

Another difference between the two approaches is that in the DPO approach, as well as a graph for the left-hand side and the right-hand side, there is also an *interface graph* that just contains the context elements. The SPO approach is simpler in that an interface graph is not required.

2.3.4 Controlling Rule Application

A user of a graph grammar system wants to be able to control when individual productions should be applied to a graph and when they should not. This is done through *application conditions*. So far, we have only discussed positive application conditions. The left-hand side of a rule defines these, so that the production can be applied when and only when there is a match for the left-hand side in the host graph.

Negative application conditions (NACs) can also be defined. These specify situations in which the rule cannot be applied. A common form for expressing negative application conditions is by a graph, or a number of graphs if a particular production has more than one negative application condition. A production can only be applied to a particular section of the host graph if there is a match for the left-hand side of the production, and if for each NAC, there is not a match for the NAC in that section of the host graph.

As an example, the production rule used in the previous section will be returned to, but this time a negative application condition will be included, as shown in Figure 2.3.4a.

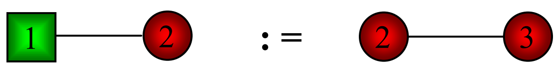
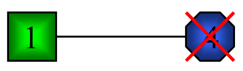
The production	The negative application condition
	

Figure 2.3.4a: A production with a negative application condition

The negative application condition specifies that the production can only be applied if there is *not* a node matching Node 4 that is connected to the node matching Node 1 via an edge. Notice that a morphism is also used here to show that Node 1 in the negative application condition is the same as Node 1 in the production.

Figures 2.3.4b and 2.3.4c show two similar graphs. The only difference is that in the graph in Figure 2.3.4c there is an edge between Node 1 and Node 3. The production above with the negative application condition could be applied to the graph in Figure 2.3.4b, but not to the graph in Figure 2.3.4c.

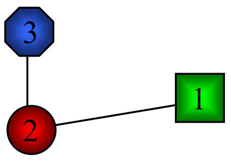


Figure 2.3.4b

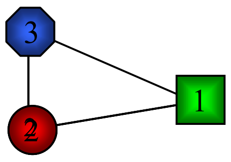


Figure 2.3.4c

2.3.5 Summary

In this section, the graph grammar approach has been described. In Section 2.3.1, it was seen that graph grammars are a type of grammar that manipulates *graphs* rather than *strings*. The two main types of graph grammars, namely the algebraic and algorithmic approaches, were then described. It was seen that in the algorithmic approach, the user has less control over how new nodes created by applying a rule are connected to the remainder of the graph, and for this reason an algebraic graph grammar will be used in this project.

Algebraic graph grammars can either employ the Double Pushout (DPO) approach or the newer Single Pushout (SPO) approach. Based on the information about these two methods, the SPO approach will be used. This is because it seems the simplest technique and it was deemed that its method of dealing with dangling edges (see Figure 2.3.3b) was the most sensible.

2.4 Graph Grammar Tools

2.4.1 Task Specific Tools

The purpose of this section is to explore some of the graph grammar tools that have already been developed, in order to assess if any of them are suitable to be used in this project as the underlying graph grammar system for the automatic Dungeon Generation System.

It was first thought that there were many tools that allowed programming using graph grammars, but this proved not to be the case. Most of the tools just *used* graph grammars and were developed to assist in a specific area. Several are tools such as DiaGen (described in [14] and [39]), created by the University of Erlangen, which are designed to allow the development of graphical editors.

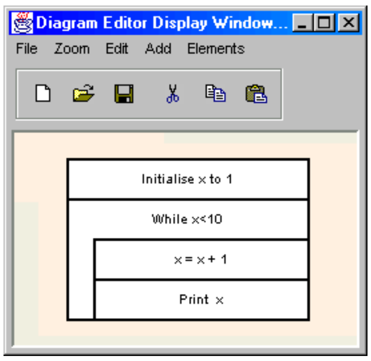


Figure 2.4.1a: Screenshot of graphical editor for Nassi-Schneiderman diagrams, created using DiaGen [63]

A tool that may be of use in this project is Treebag, developed by the University of Bremen [61]. This allows the creation of a type of graph grammar known as a *collage grammar* (described in [14]). These are context-free grammars that are used to generate pictures. Because of this, they could not be used to create the topology of a level, but may be able to be used to generate the geometry of a level.

A collage is made up of a set of parts and a number of *pin points*, which are used to insert collages into collages. Collage grammars are best explained using an example. An initial collage is specified, such as the one in Figure 2.4.1b. The four points on the edge of the collage are pin points and the structure in the centre is known as a *hyperedge*. This acts like a placeholder, and is there to be replaced by a collage using a production whose left-hand side consists of that hyperedge label.

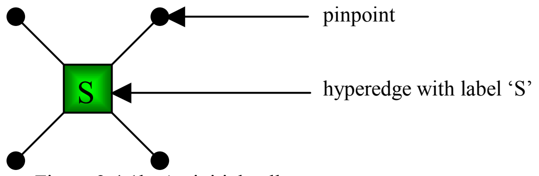


Figure 2.4.1b: An initial collage

This collage graph grammar has two productions, which are shown in Figure 2.4.1c. Both turn a hyperedge with label 'S' into a collage.

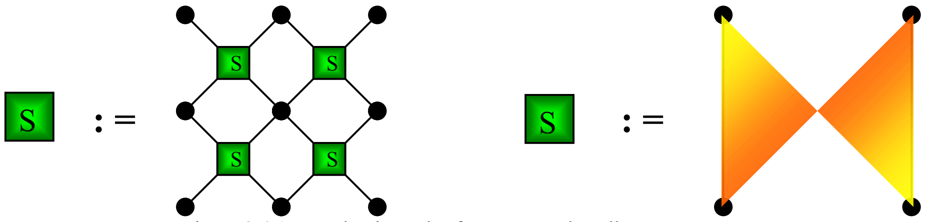


Figure 2.4.1c: Production rules for an example collage grammar

The derivation then takes place by repeatedly applying the production rules until no hyperedges are present in the collage. Choosing a production from rules with the same left-hand side (such as in above) is done at random. Some possible derivations from the initial collage in Figure 2.4.1b using the productions in Figure 2.4.1c are shown in Figure 2.4.1d: -

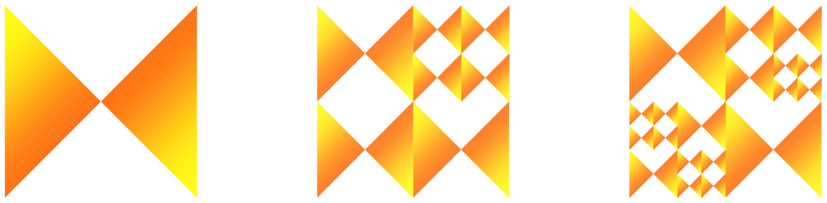


Figure 2.4.1d: Three possible derivations of the collage grammar

With Treebag, it is possible to define much more complex collage grammars, such as those that draw plants and trees. The problem with using the tool for generating levels for dungeon games is that a games company may want the level generator to generate levels in a number of stages (so they could manually check or enhance the level at each stage) and this would not be possible if the tool created a geometric description of a level directly. In addition, it would be difficult to extract information, such as the size and difficulty of the level, from what would essentially just be an image.

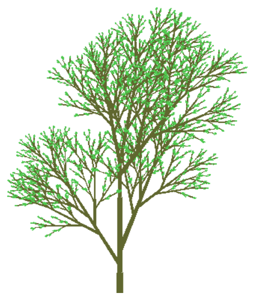


Figure 2.4.1e: Screenshot of a tree generated by a collage grammar by Treebag [61]

2.4.2 General Programming Tools

In this section, the two tools found that allow the manipulation of general graph grammars are discussed. The first of these is PROGRES [62], created by the University of Technology at Aachen. This system allows the manipulation of so-called DIANE graphs, which are DIrected, Attributed, Node and Edge labelled graphs. It was originally developed for use with software engineering tools but now has a more widespread scope.

PROGRES itself has only very limited ability at defining and manipulating attributes, but these can be programmed in C libraries and integrated into the system. Each node is given a certain type, such as Account or Customer, and all the nodes of a certain type have the same attributes, as is the case with classes in object-oriented programming. Edges also have types. These define which types of nodes the edge can connect together. Node types can be arranged in an inheritance hierarchy so that some node types can inherit attributes or integrity constraints from another node type.

An example of part of a definition of a graph grammar system is shown below. It defines a node class BIRD that ALBATROSS is a subclass of, as well as a node class COLOUR with subclasses of BROWN and WHITE. An edge with label isColour is defined that goes from a BIRD node to a COLOUR node, meaning the colour of the bird. The edge type cardinalities, such as [0:n], are used to define the type of the relationship. For example, `BIRD[0:n] → COLOUR[1:1]` means that every bird has one and only one colour, and that many birds can have the same colour. The 'intrinsic' construct defines a class attribute, as is used in the example to define the name of each bird.

```
node class BIRD end;
      intrinsic name : string;
node type ALBATROSS : BIRD end;
node class COLOUR end;
node type BROWN : COLOUR end;
node type WHITE : COLOUR end;
.. .. .. .. .. .. .. ..
edge type isColour BIRD[0:n] → COLOUR[1:1];
```

Productions are defined in the standard fashion for graph grammars, with left and right -hand side graphs, although in PROGRES negative nodes and edges, representing negative application conditions, are represented on the right-hand side graph instead of having separate graphs for each NAC.

The other general graph grammar tool is the AGG (Attributed Graph Grammar) system [60], developed by the graph transformation group at TU Berlin (see [14] and [60] for more information). The system is written in the object-oriented language Java and allows attributes of nodes to be any Java class, even user defined classes. The tool comes with graphical editors for defining nodes, edges, productions and attributes. The system allows both edges and nodes to have attributes.

AGG is based on the single pushout approach discussed in Section 2.3.3. It supports both simple and hierarchical (layered) graphs, although not all of the features available for simple graphs are available for hierarchical graphs. Each node and edge may have a label known as a *type*, and the system allows numerous edges of the same type between two nodes. This can be done because AGG uses the object-oriented approach of giving each object in the graph, such as a node or edge, its own object identity, so the value of several objects can be the same without problems occurring.

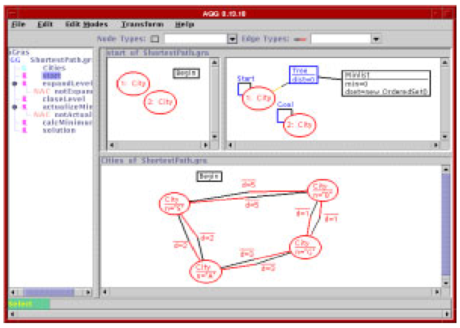


Figure 2.4.2a: Screenshot of AGG [60]

An example of a graph concerning a (greatly simplified) library system is defined visually in Figure 2.4.2b. Each box represents a node and the title in each box is the node type. Below the title, the attributes of the class are defined. For edges, the type of the edge is above the line, and below it are the attributes of the edge.

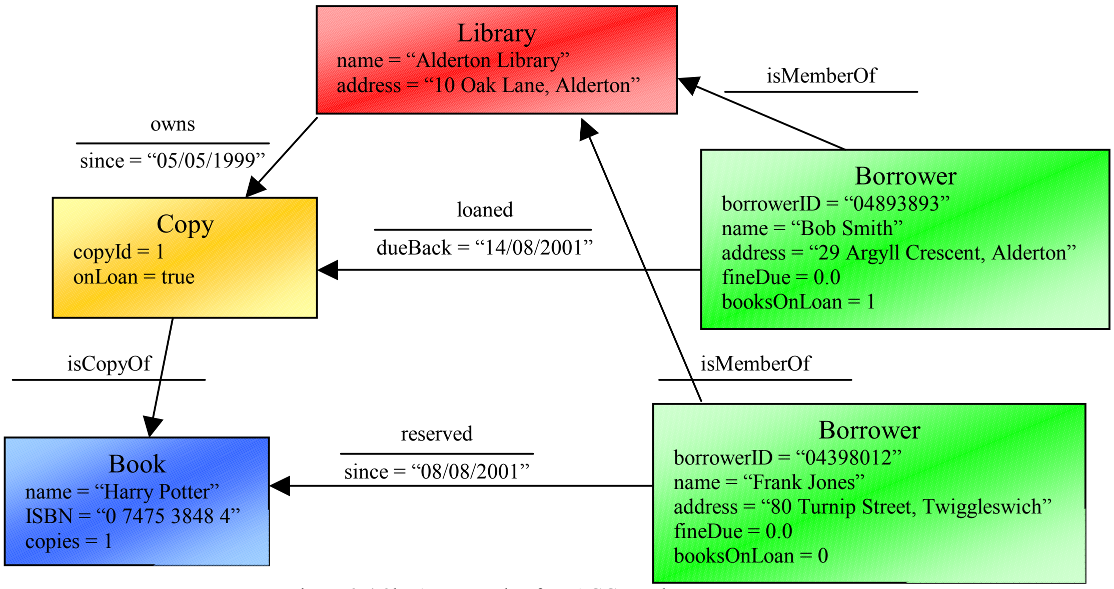


Figure 2.4.2b: An example of an AGG graph

In AGG, productions are called *actions*. They have a before state that specifies all of the preconditions (i.e. the match) and then an after state that shows the changes. On these before and after graphs, only attributes that need to be tested or whose value is altered need to be shown.

At first it may seem that both PROGRES and AGG are suitable for use in this project, but a closer examination reveals that this is not the case, for reasons relating to the control that a user of the system has over rule application. At the moment, PROGRES can only be used as a pure specification language. There are plans to extend it to give the user control over what strategy the system employs in rule application. To achieve this the developers of PROGRES plan to introduce what they have termed *pragmas* which would give the user control over what strategies were used by the system at runtime [56].

In AGG the production rules are stored in a list. A derivation is obtained by applying the first rule in the list repeatedly until it can no longer be applied. Then the second rule in the list is applied as many times as possible, and so on for all of the production rules in the list. The derivation is over when all rules can no longer be applied.

As well as not allowing different rule application strategies, the strategy used is unsuitable for use in generating dungeon levels. The system developed will need to allow parameters, such as the size of the level required, to be input and used to select those rules likely to create a level with those properties. In addition, the strategy must obviously include random elements to generate random levels. The developers of AGG say that allowing a strategy for rule application to be defined in Java is one of the next extensions they plan for the system, but such an extension will come too late for this project.

To summarise, an existing graph grammar tool has not been found that is suitable for use in the project. This is because those that presently exist can either only be used for a specific task, such as creating graphical editors, or at present do not give the user sufficient control over rule application to be useful. Due to the lack of an existing tool to use, one will have to be developed during the course of this project.

2.5 Level Design

The purpose of this section is to explore what makes a level enjoyable to play. This is being done because to be able to make the system generate fun levels and to create algorithms to assess this, this knowledge is needed. Two techniques are examined to see what they can tell us about making levels enjoyable, and the section ends by looking directly at the thoughts of game developers on good level design.

2.5.1 Game Theory

Put simply, game theory is the study of trying to understand how to choose rationally between several alternatives. It was created by von Neumann and Morgenstern in 1953 [64] and has since been used widely in subjects such as evolutionary biology ([32], [43]), economics ([24], [33], [48]) and philosophy ([47]). Excellent general introductions to game theory can be found in [33] and [59].

Game theory models *contests* between two or more individuals known as *players*. In a contest, each player can gain benefits from the contest and sustain costs. The *payoff* of a contest for an individual player is the overall consequences for them, the benefits gained minus the costs sustained. The aim of players in contests is therefore to maximise their payoff.

The players have open to themselves a finite number of strategies they can use. Each player chooses a strategy without knowing the strategy chosen by the other players, and for each combination of strategies there is a certain payoff to every player. Two of the assumptions in game theory are that all players know all of the strategies that they and the other players can use, and that each player acts rationally, choosing the strategy that will give them the greatest payoff.

An example that will help to get across the main points of game theory is now presented. The Prisoner's Dilemma is perhaps the best-known game theory problem and was first presented at a lecture in 1950 by Albert Tucker [33]. It concerns two prisoners who have carried out a burglary together and have been caught by the police. They are taken to separate cells and are interviewed. If they both confess they will get five years in jail, if neither of them confess they will be sentenced on a minor charge and face one year in jail, and if only one of them confesses (and implicates the other) then that man shall be set free and the other will have a sentence of ten years.

In this game, each of the prisoners is a player with two strategies open to them – confess or do not confess. Their sentences can be thought of as the payoffs. Each of the combinations of strategies is shown in Table 2.5.1a, which is known as a *payoff table*. The number to the left of the comma in each box is the payoff to Prisoner 1; the number to the right is the payoff to Prisoner 2. All of the payoffs are negative because they will not gain from the contest, whatever happens.

		Prisoner 2	
		Confess	Do not confess
Prisoner 1	Confess	**-5, -5**	**0, -10**
	Do not confess	**-10, 0**	**-1, -1**

Table 2.5.1a: Payoff table for the Prisoner's Dilemma

So, what will the prisoners choose to do if they act rationally? Each prisoner should think as follows: -

- If the other prisoner confesses, then it is best for me to confess (-5 rather than –10).
- If the other prisoner does not confess, then it is still best for me to confess (0 rather than –1).

Therefore, both prisoners will confess and each will get five years in jail. This is an example of where rational thought trying to maximise an individual's payoff can lead to everyone being worse off.

In the early 1950s, John F. Nash published several papers ([40], [41]) that revolutionised game theory by introducing the concept of *Nash equilibria*. A Nash equilibria is a set of strategies used by players such that no player can improve his or her payoff by changing their strategy. The Prisoner's Dilemma exhibits a Nash equilibrium, because when both prisoners confess, no player can improve his payoff.

How does game theory relate to levels of dungeon games? At the beginning of this section, it was said that game theory related to games with two or more players, and the levels that will be generated by the system are to be designed for single player use. However, a branch of game theory, called optimality theory or decision theory, applies to single-player games. It concerns choosing a strategy from uncertain alternatives. Each has a payoff and a probability of obtaining that payoff.

It is easy to think of ways in which decision theory is applicable to levels. For example, when designing a level, the designer wants to make sure that any parts of the level that do not strictly need to be visited to complete the level are still visited by the player. For this to be the case, the player must perceive, through experience of previous levels, that the rewards gained by exploring that area are greater than the costs. In other words, the designers must make sure that the payoff to the player of choosing the strategy of exploring these areas is greater than the payoff of avoiding them. This was often not the case with the levels generated by Simon Ince in his dissertation, which he cited as one of the main faults of the system.

For instance, if the only rewards were health packs, then the expected loss of health to the player in battling past the opponents in the area must be less than the benefit they obtain from the health packs they receive once having passed the opponents. If this is not usually the case, then the player will choose the strategy of not exploring these areas, affecting their overall gaming experience.

2.5.2 Psychology and Game Design

In this section, it will be examined what can be learnt from different branches of psychology in order to design good levels for dungeon games. The first of these branches is behavioural game design (see [17] for a more detailed account). Unlike game theory, which predicts what a person should rationally do in a situation, behavioural psychology can give us an idea of what they might actually do.

Its relation to game design is in the area of how people (or indeed animals) react to rewards. These rewards, such as finding some health packs or finishing a level, are known as *reinforcers* in behavioural psychology. The rules dictating when reinforcers are given to the player (such as when they reach a certain part of the level) are known as *contingencies*. A *response* is the term used when a player acts to fulfil a contingency, such as reaching a certain area of a level or getting past an opponent.

If a player is used to receiving a reinforcer for performing a particular response, then if this stops being the case (i.e. no reward is given) then this can leave the player frustrated, and possibly cause them to quit the game. This concept is called *extinction*. An example of this is the previously discussed sections of a level that are only visited to obtain rewards. If after a while, the player came to one section, spent a while getting past opponents to find there was no reward at the end and he or she had wasted their time, then they would feel annoyed.

Figure 2.5.2a: A common reinforcer used in Dark Forces is the extra ammunition gained after killing an opponent (Screenshot from *Dark Forces* [25])

Another effect that should be avoided in a game is *behavioural contrast*. This is when the size of a reinforcer is increased, and then drops back down again on a subsequent occasion. The larger reinforcer raised the player's expectations and they are no longer satisfied with the smaller one. Examples here are having a large drop in the number of health packs available between levels or introducing a more interesting type of opponent on one level only to go back to the less interesting opponents on previous levels. It also applies to a sharp increase in the difficulty of levels.

To keep players playing a game, there should be a high activity level with constant rewards to keep the player interested. In dungeon levels this can be achieved by constantly letting the player explore new areas of a level, as reaching a new area is a reward in itself.

The short-term memory of a human can normally only store seven plus or minus two items at any one time [37], and this can cause problems in games, such as *exploration anxiety* [13]. If levels are not designed carefully and all parts of them look similar then when the player explores them they may get lost, forget where they have been, and become frustrated.

2.5.3 Good Level Design

In this subsection, some direct ideas from game designers on what makes a good level will be presented. These have been grouped into the following categories: -

Interesting level architecture – dungeon games are full of mazes – twisty passages with rooms here and there. However, this has been done so often that such mazes are often boring ([1], [53]) and players can easily get lost in them because different areas look so similar. They can be made more interesting by varying their size and shape, and by making use of other components such as open areas, staircases and landings, that are seen as places where battles can take place, not just passages from one room to another [9]. Levels should also not have pointless areas that serve no function [53].

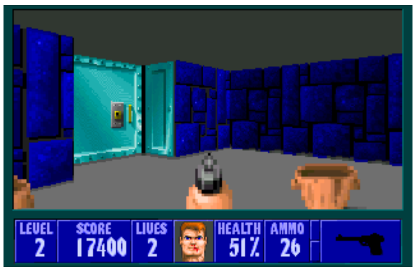


Figure 2.5.3a: Empty rooms and completely flat levels contributed to Wolfenstein levels being unexciting compared with future dungeon games (Screenshot from *Wolfenstein 3D* [18])

Pacing – some designers recommend that the player be in constant fear of his character dying (through constant opponents) to keep the player constantly alert [54], while others believe that such a constant fear can 'dull the senses' [6] and so there should be sections were no opponents are present.
Realism [1] – a game is more enjoyable when the level is believable. By this, it is meant that the setting of the level could actually exist taking into account the story. For example, a level just full of empty rooms with monsters walking around randomly cannot add to the storyline and may just leave the player thinking 'What is this place?" or "What use could this building possibly have?"
Intelligent opponents [1] – even through advances in artificial intelligence, most of the opponents in games are stupid. They employ no tactics and just move towards the player's character, shooting. Game designers often use sheer numbers of opponents to increase the difficulty of a level rather than having fewer, more intelligent opponents.
Landmarks [54] – landmarks are features of a level that stand out, such as a large open space, a tower or a distinctive building. They both add character to a level and allow the player to explore a level more easily by acting as reference points.

Figure 2.5.3b: An example of an easily recognisable landmark in Half Life (Screenshot from *Half Life* [57])

Non-linearity ([53], [54]) – non-linearity is all about having more than one way to overcome a challenge or solve a level. This could be achieved in several ways, such as giving the player alternative routes to get to the end point of a level or allowing them to do things in any order they wish. Some non-linearity is beneficial because players feel like they have more freedom and control over the game, and they are more likely to manage to complete a level and so continue playing the game if there are multiple ways a level can be completed.
Careful asset revelation ([52], [54]) – the 'assets' in a game are things such as the types of monsters and weaponry that can be put in levels. Asset revelation is the process by which these assets are first used in a game, by introducing them into a certain level. New assets that the player can use or fight against help to keep the player interested in playing the game. They should not all be used on the first level, but should be introduced throughout the whole game.
Resource balance [6] – extra ammunition and health packs should be placed to achieve a balance where the player is always close to running out of these commodities, but rarely does (assuming that the player is skilled enough at the game).
Difficulty [52] – if a level is so difficult that few players can complete it then this may lead many people to stop playing the game altogether. Conversely, if levels are too easy and are not challenging then people may also stop playing the game. The difficulty of a level must therefore be between these two extremes. In addition, as the player gets better at a game, the later levels should reflect this by being more difficult.

Not all of the above items can be tested for in the Dungeon Generation System to be developed. For instance, the system will just place opponents in levels, not determine how intelligent they are. However, most of the items are relevant and will be considered when assessing how much fun a level is and in generating levels.

2.6 Summary

To summarise, it has been seen that dungeon games, or first-person shooter games as they are more widely known, are a specific genre of computer game that is mainly concerned with exploring a three-dimensional world while killing opponents. The topology and geometry of a level were defined to distinguish between descriptions of a level detailing in what order things were encountered and an actual physical description of a level's layout.

The aim of this project is to generate topological descriptions of levels that capture such properties of dungeons games as the order of play and the location within that order of level objects such as opponents, health packs, keys and doors. The exact physical placement of objects or room dimensions will not be described by the system; neither will graphical issues such as how the level objects will look or the architecture of the level.

Previous ways of generating game content automatically were discussed, namely fractal and bitmap terrain generation, as well as the work done by Simon Ince using context-free grammars. It was seen that there were a number of problems with Ince's level generation system. These included levels being produced that were too similar to each other, that levels were often produced that were not close to the size and difficulty required and that the system could not represent complex corridor designs.

In Section 2.3, a type of grammar known as graph grammars was introduced, and it was seen that graph grammars provide a more natural structure than string grammars with which to model the complex connectivity of levels. They also had the advantage of being more flexible in terms of controlling the process of rule application, which will allow levels to be produced that more closely match the properties required by the user. Based on these advantages, graph grammars will be used as the method to generate levels in this project.

Next, tools for using graph grammars were examined and it was found that none of the tools available gave sufficient control to be useful to the project. Perhaps the main conclusion that can be drawn from this chapter is therefore that such a generic graph grammar system will have to be developed before thinking about generating dungeons.

Lastly, game theory and psychology were examined to see what could be learned from them about good level design. The conclusions drawn from this were that they both had information that would help to make levels more fun to play. For example, the level of rewards should be sufficiently high to encourage the player to visit all areas of the level, and to avoid player disappointment and frustration the level of rewards at a particular place should be proportional to the time and effort it has taken the player to reach those rewards.

A number of factors were also listed that current game developers consider improve the quality of a level, such as having interesting level architecture and non-linearity. As will be seen later, not all of these factors (such as the intelligence of opponents) can be estimated for the levels that the system will generate, but those that can, will be used to try to assess the fun-value of the levels.

Chapter 3 – A General Graph Grammar System

As explained in Chapter 1, the Dungeon Generation System will need a general Graph Grammar System to support graph transformation. It was found in Chapter 2 that the presently existing graph grammar tools were not sufficiently flexible, and so in this chapter the design and implementation of such a system is described. The chapter ends with findings into the power of context-free graph grammars, which was discovered during the testing of the Graph Grammar System.

3.1 Design

3.1.1 Specific Requirements

The Graph Grammar System should be totally independent of the Dungeon Generation System, in that it should not say anything about concepts such as dungeons, levels, difficulty or size, just general nodes and edges. However, the system must have certain features to enable it to be used to generate levels. These are listed below: -

- Allow probabilities to be associated with productions – this is desirable because it allows the user to specify that some rules should be more likely to be chosen than others.
- Allow flexible matching conditions – productions should be allowed to contain arbitrary tests to see if they can be applied.
- Allow any strategy to control rule application to be used – if the Dungeon Generation System is to have strategies to generate levels of a certain size and difficulty, then the system must allow other strategies to be used.
- Have a default strategy that selects and applies productions randomly – all graph grammar systems must have a strategy to use if the user does not want to supply one. This default strategy should be random so that if the user wants a level generated and does not specify any constraints then this strategy can be used.

3.1.2 Graph Grammar Approach and Language

After considering the different types of graph grammar approaches upon which the system could be based, it was decided in Section 2.3.5 that the Single Pushout Approach (SPO) would be used. There are, however, further decisions to be made in regard to the graph grammar model to follow.

To increase flexibility, the system will allow directed edges as well as undirected edges, although it remains to be seen whether directed edges will be necessary for the generation of levels. It was decided that an attributed graph would be used, as in the AGG system (see Section 2.4.2). This will allow concepts such as the number of opponents in a particular place to be expressed much more succinctly. However, it was decided that allowing edges to have attributes was unnecessary. This is because, in relation to game levels, edges will be used purely to indicate which nodes can be accessed from one another and will therefore not need to store level data.

The system will be written in the object-oriented language Java with the whole system forming a Java package called ggs (Graph Grammar System). This will allow the flexibility needed for matching conditions because in a Java method anything can be written. The use in AGG of any Java object to represent an attribute is seen as a very desirable feature and one that will also be used in this system.

Furthermore, the inheritance allowed in Java will allow new node types and strategies to specialise existing ones. Methods to create nodes or apply strategies can then take as arguments the most general class possible, such as Node or Strategy, allowing an object of any class extending these to be passed in. This will make sure that any type of node or strategy can be created by the user and used in the system.

3.1.3 Class Design

As would be expected in an object-oriented system, there are classes in the system to represent concepts such as a node and an edge. The main classes in the system and how they relate to each other are listed in Table 3.1.3a.

Name	Features	Purpose
ComponentProduction	abstract	Represents a component production.
DefaultStrategy	extends Strategy	The default strategy used by the system to select which production will be applied next.
Edge		Represents an edge between two nodes. Has references to these nodes through source and target attributes, as well as an edge type. Static constants `UNI_DIRECTIONAL` and `BI_DIRECTIONAL` are defined here representing the edge types.
Graph		A graph with a collection of nodes and edges. Has methods to add and delete nodes and edges, as well as checking for matches and applying a production.
Morphism		Is used to map between nodes in a production and those in the host graph when a match has been found.
MultiProduction	extends Production abstract	Contains several ComponentProduction objects.
Node		Represents a node. Each node has a unique ID and a type, which determines if two nodes match. Is extended by node specialisations.
ProbabilityChooser		Selects an object randomly where each object can have a probability associated with it.
Production	abstract	Represents a production. It contains those methods that are needed in both the SingleProduction and MultiProduction classes, such as returning a match and seeing if the attributes match those in a section of the host graph.
SingleProduction	extends Production abstract	Represents a single production.
Strategy	abstract	An abstract class that actual strategies can extend by implementing the apply() method.

Table 3.1.3a: The main classes in the ggs package

A few points about the above classes will now be clarified, mostly those concerning productions. In the Graph Grammar System, there are two types of production that can be used. The type that will generally be used (by extending this class) is SingleProduction. This represents a simple production with a left-hand side and a right-hand side.

The productions can be supplied to the strategy to be used in a ProbabilityChooser object. It is this that allows the user to specify that certain productions should be applied more often, as a probability is associated with each production. This probability is simply a number. A strategy can then request the ProbabilityChooser to return a production based on the probabilities associated with each production.

The second type of production is a MultiProduction. To see how they can be used, consider the three productions in Figure 3.1.3b, which all have one node of type 'S' as their left-hand side graph.

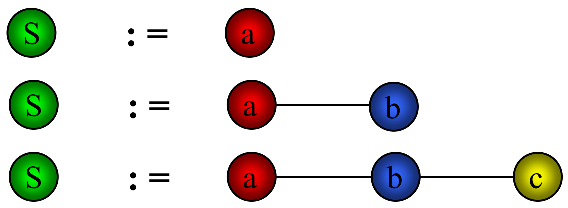


Figure 3.1.3b: Three productions with the same left-hand side

A MultiProduction would eliminate the need for the left-hand side to be repeated by defining the left-hand side once and then having three ComponentProduction objects that specified one of the right-hand sides each. Probabilities could be associated with each of the three ComponentProductions to make some of the transformations more likely than others. MultiProductions are not strictly necessary but can be used to avoid repetition, especially where the left-hand side is complex with many nodes, edges and attribute tests.

3.1.4 Algorithm Design

Two of the algorithms developed for the system are complicated enough to warrant an explanation of how they work. The first of these is the apply() method (and its subcontractor methods) of the Graph class that sees if a production can be applied to itself and then applies it if this is the case.

For grammars that deal with strings rather than graphs, looking for a match is a simple task. In a leftmost derivation, the leftmost nonterminal in the string is expanded next and in a rightmost derivation the opposite occurs. However, graphs are not linear; they have no start and end points, and the nodes are in no particular order. In addition, although a context-free match requires only instances of one node to be found, context-sensitive matches require a whole subgraph to be matched. Therefore, finding matches for graph grammar productions is not a trivial task.

It was investigated how the two general graph grammar tools PROGRES and AGG (see Section 2.4.2) implemented graph matching. Neither gave a detailed description. AGG performs graph matching by treating it as a constraint satisfaction problem (CSP), which is the problem of assigning values to a set of variables, where a number of constraints must be met (see [51] for more on CSPs).

PROGRES performs graph matching in a number of stages [56]. Firstly, it finds matches for *simple node patterns* and *edge patterns*, which are PROGRES terms for node and edge instances on the left-hand side of a production. Finally, it deals with Negative Application Conditions. It does this by taking each *negative node* or *edge pattern* (nodes and edges that must not exist in the matching subgraph) at a time and tries to extend the matched subgraph with a match of the negative node or edge. Failure at any of these stages causes PROGRES to backtrack so that it can look for another match within the host graph.

The graph matching algorithm developed here uses the basic idea used in PROGRES. However, because it is desirable for it to randomly match a rule, it finds all possible matches for a rule in the host graph and then selects one at random, rather than just returning the first match that it finds.

Firstly, it finds all combinations of nodes that matched, ignoring whether the required edges were present. A list of candidate matches was kept. This started off as all the possible matches for the first node (based on some order) and then grew or shrunk as each of the subsequent nodes in the production tried to match with other nodes in the host graph, taking into account those already matched in each candidate match.

The candidate matches were then filtered, based initially on whether the edges required between the matching nodes were present. For example, if the production to be applied is that in Figure 3.1.4a and the graph is that in Figure 3.1.4b, then before edges are considered there will be two candidate matches:

a) Node 1 matches with Node 4, Node 2 matches with Node 5.
b) Node 1 matches with Node 4, Node 2 matches with Node 6.

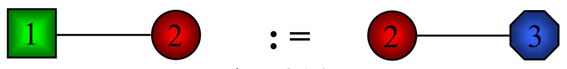


Figure 3.1.4a

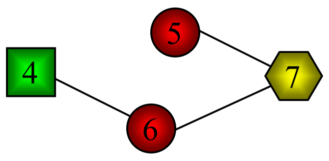


Figure 3.1.4b

However, only b) is valid when edges are taken into consideration, so Candidate Match a) would be filtered out at this stage leaving only one candidate match.

The filtering process is repeated for any attribute tests and negative application conditions that were specified by the production. Out of the remaining candidate matches, one is chosen at random and the production is applied using that match. If no candidate matches are left then the production cannot be applied. By considering all possible ways in which a production can be applied and then choosing one randomly, the way a graph is expanded by a certain production will be different every time, increasing the random aspect of the process.

The other algorithm worth mentioning is the apply() method of the DefaultStrategy class. This method basically executes the default strategy. The default strategy has been designed to be as random as possible. This is achieved by choosing a production randomly out of the list of all productions until one is found that can be applied.

As well as the productions to be used and the start graph, the method can also be passed as arguments a Vector of nonterminal nodes, and two integers called `attempts` and `averageDerivations`. The nonterminal nodes are any nodes that the end graph may not have in it, and are just used to represent higher-level structures that will be rewritten to terminal nodes by further derivations. In string grammars, the derivation ends when all symbols in the string are terminal symbols, but this is not the case in graph grammar derivations.

The `averageDerivations` variable gives the user some control over the number of derivations to be applied before stopping. The number passed as a parameter is multiplied by a random number following a Gaussian distribution to produce the number of derivations that the strategy will try to stop after. This may not be the actual number of derivations because if at this point the graph still contains nonterminal nodes then it will need to be expanded further.

Sometimes, derivations may be chosen that stop a graph ever being reached that contains no nonterminal nodes. Consider the two extremely simple productions in Figures 3.1.4c and 3.1.4d. Assume the start graph is a single node of type 'S' and that nodes of type 'S' and 'E' are nonterminal nodes. If Production 2 is chosen instead of Production 1 then the graph will just contain one 'E' node. No productions are now applicable and the graph is unacceptable because it contains a nonterminal node.

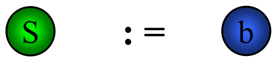


Figure 3.1.4c: Production 1

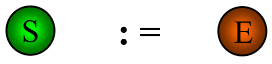


Figure 3.1.4d: Production 2

Thus, the system must start the derivation again from the initial graph. The `attempts` variable is used to control how many times the strategy will try again. After this number of times, it returns failure. Another approach that could have been used is backtracking. Here the system would store past graphs as well as details of which rules have been applied, so that it could revert to a previous graph if it got stuck and try to apply a different rule. However, this approach has not been used for the reason given below.

Nonterminal nodes are used far less frequently than nonterminal symbols are in string grammars. In fact, most graph grammar systems do not even possess the concept of nonterminal nodes. Therefore, in practice there would be few occasions when the system actually did get into a situation where nonterminal nodes were present and no rule could be applied. Backtracking would slow the system down due to all of the extra information that would need to be stored throughout the derivation, and it was felt that the benefit due to backtracking would be outweighed by the extra time overhead.

Besides, this is simply the way the apply() method of the DefaultStrategy works. A different Strategy object could be created that behaves differently, and in Chapter 4 it will be seen that the Dungeon Generation System uses its own Strategy classes, which implement a type of backtracking by storing several intermediate graphs that can be returned to if failure occurs.

3.2 Example

A sample output from the system is shown in Figure 3.2a. The current output from the system is textual, but is adequate for debugging needs and to demonstrate that the system is working. The initial graph is a single 'Start' node, and there are three productions, which are shown in Figure 3.2b (see the key in Figure 3.2d). The derivation is also shown graphically in Figure 3.2c. The rules generate graphs consisting of zero or more 'Room' nodes connected in a line.

To summarise, the Graph Grammar System is a system that is designed to allow the manipulation of graphs. It is general in the sense that it says nothing about dungeons or levels and could therefore be used for many different graph grammar applications. It contains classes such as Node and Edge to represent the basic components of all graphs, as well as abstract classes such as Production, which is extended to create classes to represent the different production rules in a particular graph grammar application.

The system has been developed so that it contains all of the features (that were not present in existing graph grammar tools) that allow it to be used as the foundation for constructing graph grammars to generate levels for games. These include the fact that the system randomly chooses a context in which to apply a rule (out of all the possible ways a rule could be applied to a graph), and giving the user full control of rule application by allowing them to define their own strategies that determine how rule application proceeds.

```
****Start****Attempt 1*********

Nodes:
Start1
Edges:


Nodes:
Temp8
Room9
Edges:
Room9--Temp8


Nodes:
Temp8
Room9
Room10
Edges:
Room9--Room10
Room10--Temp8


Nodes:
Room9
Room10
Edges:
Room9--Room10
************End************

Nodes:
Room9
Room10
Edges:
Room9--Room10

Succeeded
```

Each of these sections represents the state of the graph after a derivation

Each node has a unique ID after its type, so that it can be seen which nodes are connected together.

Figure 3.2a: Sample output from the system

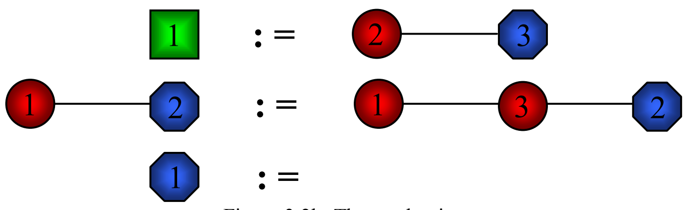


Figure 3.2b: The productions

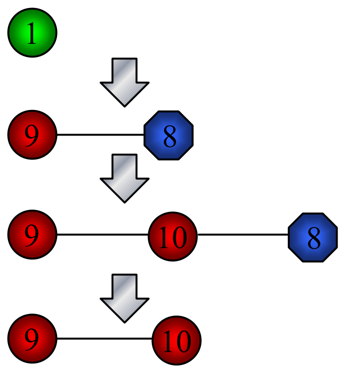


Figure 3.2c: The derivation

🟩	‘Start’ node
🔵	‘Temp’ node
🔴	‘Room’ node

Figure 3.2d: Key to node types

3.3 The Power of Context-Free Graph Grammars

For a graph grammar to be context-free, all of the productions must only have a single node on the left-hand side. Therefore, the graph grammar in the example of Section 3.2 is not context-free because the second production in Figure 3.2b has two nodes on its left-hand side.

Could the graph grammar in Section 3.2, which is after all generating very simple graphs, be altered so that it was a context-free graph grammar? Two attempts at this are shown in Table 3.3a, with the productions given first, followed by a graph produced by them showing why it fails to produce a line of ‘Room’ nodes.

In fact, it is impossible to generate such graphs with context-free productions. This example demonstrates that the set of graphs that can be generated by context-free graph grammars is much smaller than the set that can be generated by context-sensitive graph grammars.

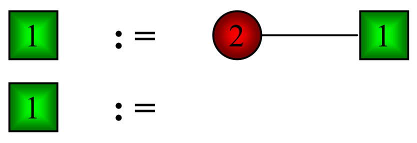
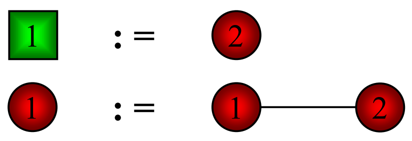
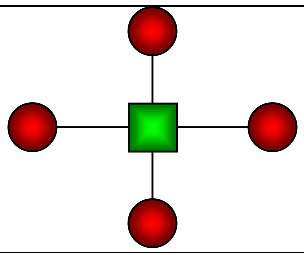
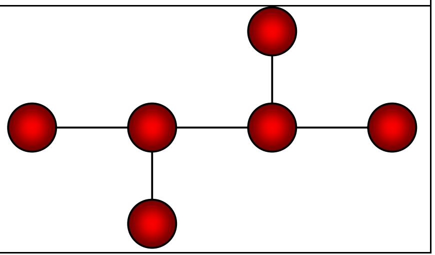
	Attempt 1	Attempt 2
Productions		
Example showing why it does not work		

Table 3.3a: Failed attempts at making a context-free graph grammar produce the linear ‘Room’ node structure

One of the main factors causing this is what will be called the *connection problem*. The connection problem is that unless the node on the left-hand side of a production also occurs on the right-hand side, then it is impossible to add new nodes that are connected to the rest of the graph. This is the case in Figure 3.3b where a context-free production has a node on the left-hand side that does not feature on the right-hand side. When it is applied to the graph in Figure 3.3c, the nodes matching Nodes 2 and 3 become unconnected from the rest of the graph.

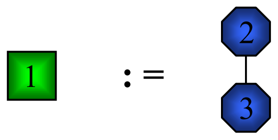


Figure 3.3b: The production

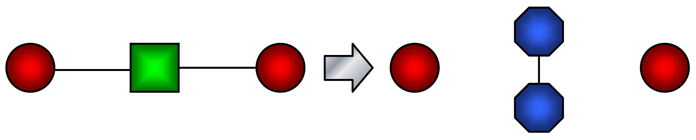


Figure 3.3c: Applying the production

To summarise, unlike context-free string grammars, which do not require edges to connect the symbols together, context-free graph grammars are much less powerful that their context-sensitive counterparts. However, in some other branches of graph grammars, context-free productions are more expressive. For example, many complex images can be generated using the context-free collage grammars described in Section 2.4.1. In these collage grammars, edges are not needed due to the pictorial nature of the grammars.

The conclusions that can be drawn from this result are as follows. Firstly, it backs up the decision made to provide support for context-sensitive matching in the Graph Grammar System, because without this feature the system would be nearly useless, as few languages of graphs could be generated using it. Secondly, it shows that any set of rules to generate levels for a particular game must almost certainly include some or many context-sensitive rules, in order to limit the language of the grammar to those graphs that represent valid and fun levels.

Lastly, the limited power of context-free graph grammars highlights a disadvantage of graph grammars as opposed to other grammars. Context-sensitive matching is much more costly in terms of time, and because graph grammars need this feature to be of much use, they have a higher time complexity (usually exponential [44]) than other types of grammar.

Chapter 4 – The Dungeon Generation System

4.1 A Basic Strategy To Generate Levels

Now that a system that allows the manipulation of graph grammars has been produced, the next stage is to use this foundation in order to create the Dungeon Generation System. This system will be input with rules for a particular game, as well as parameters detailing characteristics that the generated level should possess, and should output a suitable graph representing a level.

To take the rules and parameters as input and deliver such a graph, the system needs a *strategy*, and this section is concerned with the initial strategy developed for the Dungeon Generation System. A short review on search concepts, which play a significant role in the working of the Dungeon Generation System, is firstly presented. Then the design of this Dungeon Generation System is described, followed by an example of it producing a level based on the values of certain parameters that have been input by the user.

4.1.1 Search Concepts

This section presents a short summary of some of the major concepts used in search algorithms that will be needed to help understand later sections. A good introduction to search can be found in [46]. The first concept that needs defining is *state space*. This is simply the set of all possible states that a particular problem could be in, together with information on which states can be reached directly from each other. A *successor* of a state s is a state that can be reached from s. State space can be represented as a graph with the states as nodes and the edges representing allowed moves between the states.

For example, consider the problem of travelling between two different towns on an island (see Figure 4.1.1a). There will be one state for each town, representing the town that the traveller is currently at, and each edge will correspond to a road between two towns, so the state space will be that shown in Figure 4.1.1b.

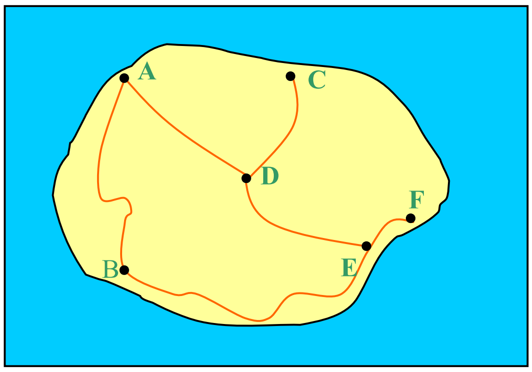


Figure 4.1.1a: Towns and the roads between them

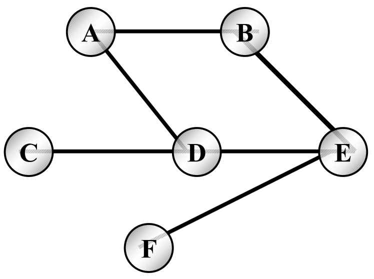


Figure 4.1.1b: The state space for the route problem

In a search, we start in a particular state called the *start state* and the problem then is to reach a *goal state*, which represents a solution to the problem. For the example, if a person wanted to travel between B and F then B would be the start state and F would be the goal state. Often, we want to know the *path* that has been taken through search space to reach a goal state. This is expressed simply as the set of states that need to be traversed to get from the start state to the goal state. In the example journey above, one possible path would be [B, E, F], although there is more than one way of travelling between the two towns.

The last concept that needs exploring is the *Dynamic Programming Principle* (see [12] and [34] for more information). In dynamic programming, problems are solved by combining the solutions to subproblems. It is applicable to most searching problems and is especially useful for optimisation problems, such as finding the shortest path.

For dynamic programming to be applicable to a problem, it must have an *optimal substructure* [34]. This means that an optimal solution for a certain problem contains within it optimal solutions for subproblems. For example, for the route problem in Figure 4.1.1a, the optimal route between B and F is simply the concatenation of the optimal routes between B and E and between E and F.

Due to the optimal substructure property, subproblems only need to be calculated once when solving a problem by dynamic programming. This means that for search problems that can be solved by dynamic programming, only the shortest route between any two states needs to be stored [34]. This means that state space can always be represented as a search tree, which grows as the state space graph is gradually explored.

4.1.2 Design

4.1.2.1 General Design

Before discussing the design of the strategies developed, the design of the entire Dungeon Generation System (DGS) will firstly be described. Like the Graph Grammar System in the previous chapter, the DGS has been implemented as a Java package (called dgs), which imports classes from the general graph grammar package in order to function. It contains all classes specific to level generation, but no examples of actual rules or nodes for a particular game, as these should be defined outside of the package.

The DGS must provide strategies that allow parameters for the size, difficulty and fun-value of a level to be input. It should then output a level in the form of a graph which is of the size, difficulty and fun required. The system therefore clearly needs some way of measuring size, difficulty and fun-value, and so the level-generating strategy will take as an argument a set of ParameterEstimator objects, each of which examines the current level for a single parameter (such as size) and outputs its estimate of the value of that parameter.

To make the system as useful as possible, although it will come with its own strategy, a user of the program should be able to write a strategy of their own that is specific to a particular game and use this when generating levels. The same applies to estimating the value of parameters. It is likely that a ParameterEstimator written with a particular game in mind would produce more accurate estimates than any generic ParameterEstimator, and therefore the system will allow users to write and pass in their own ParameterEstimator objects to the system. Information on estimating the size, difficulty and fun-value of a level can be found in Section 4.2.

4.1.2.2 Class Design

The classes in the DGS that concern the level-generating strategy and how they relate to each other are listed in Table 4.1.2.2a.

Name	Features	Purpose
LevelStrategy	abstract	Extensions of this class are strategies that are used to create a level.
LevelStrategy1	extends LevelStrategy	The simplest strategy that the Dungeon Generation System supplies in order to generate a level based on the values of certain parameters.
Parameter		Represents a parameter, such as the size of a level. Each parameter has a name indicating what it represents, a ParameterEstimator object, and the minimum and maximum acceptable value of that parameter that the level produced must have. The target value is simply the mean of the minimum and maximum values.
ParameterEstimator	abstract	Extensions of this class are used to estimate the value of a particular parameter. Any extending class must implement the estimateValue() method which takes a graph and returns a value for that parameter.

Table 4.1.2.2a: The main classes in the dgs package concerning the level–generating strategy

4.1.2.3 Algorithm Design

LevelStrategy1 extends LevelStrategy by providing an implementation for the apply() method that is called to apply a strategy to a graph. Therefore, it is this method that decides which rules to apply in order to produce a level of the required size, difficulty and fun-value. The apply() method of LevelStrategy1 is now explained in detail.

Like the DefaultStrategy class, which was described in Section 3.1.4, the strategy takes as parameters an initial graph, the game-specific productions, a set of nonterminal nodes that must not be present in the final graph, and an integer `attempts` that specifies how many times the method should attempt to produce a suitable level before terminating with failure.

However, in addition, the class is also input with a list of Parameter objects, each one representing a quality of the level that the user wishes to constrain. Thus, if the user was only interested in producing a level with a size between 50 and 100 and did not care what the difficulty or fun-value of the level was, the list of Parameter objects would just contain one Parameter object, designed to measure the size of a level. In addition, because the system does not specify that the parameters need to represent any qualities in particular, levels could be generated taking other factors into consideration apart from size, difficulty and fun-value, such as the connectivity or non-linearity of a level.

It is obvious that conflicts could occur between multiple parameters, in that given the rules it is impossible to produce a level satisfying all parameters. For example, consider if the ParameterEstimator objects for size and difficulty always produced the same output. Then, if a size of between 10 and 50 and a difficulty of between 60 and 100 were asked for, it is clear to see that the system would fail to produce a level that satisfied these parameter values.

Due to this, parameters are prioritised. If all parameters cannot be satisfied, the system will output a level that matches the top n parameters that can all be satisfied by a single level. What this means is that for the example parameter values given above, since both cannot be satisfied at once, a level would be output just satisfying the size parameter (assuming this rather than the difficulty parameter was the first in the list) and the system would signal that it had been partially successful.

The algorithm used is based on one known as *hill climbing*, a good description of which can be found in [46]. It is a heuristic approach that can only look ahead to the surrounding states in state space, which in this system are the graphs that can be produced from the current graph G by applying one production rule to G. It was chosen to base the algorithm on hill climbing because it is a simple technique with few memory overheads and it is easy to alter the technique to make it work randomly.

In hill climbing, a rule is chosen and applied, and the resulting state n (in this case a graph) is examined. If the value of the new state in terms of some parameter is closer to the target value than for the previous state, then the current state is set to n, else a different rule is applied to the current state and the value of that state is compared with that of the current state. The process repeats until either the value of the current state is the target value (or in the range of acceptable values), or until all states that can be moved to from the current state have a value that is no nearer the target value than for the current state.

The algorithm used is described more formally below: -

```
Set the current parameter to the first parameter in the list of parameters.
Set $P_{COPY}$ equal to the set of production rules $P$.
WHILE true
        Estimate the value of the current parameter for the current graph $G$, using the supplied
        ParameterEstimator object, to obtain an estimate $e$.
        IF $P$ is not empty THEN
                Choose a production rule $r$ at random from $P$.
        ELSE
                IF some parameters have been satisfied THEN
                        Exit with partial success.
                ELSE
                        Exit with failure.
                END IF
        END IF
        Create a copy $G_1$ of the current graph $G$.
        Apply rule $r$ to transform graph $G_1$.
        Estimate the value of the current parameter for $G_1$ to obtain an estimate $e_1$.
        IF $e_1$ is in the range of acceptable values for the parameter THEN
                IF the current parameter is the last parameter THEN
                Exit with success.
            ELSE
                Set the current parameter to be the next parameter in the list.
            END IF
    ELSE
            IF $e_1$ is closer to the target value for the current parameter than $e$ THEN
                Set the current graph $G$ to graph $G_1$.
                Set $P$ equal to $P_{COPY}$.
            ELSE
                Discard $G_1$.
                Remove $r$ from the set of production rules $P$.
            END IF
    END IF
END WHILE
```

However, if implemented purely as shown above, the algorithm would sometimes fail to find a solution where one existed. Consider an initial graph consisting of a single node that has an attribute `size`, which is an integer of value 0. There are two production rules, as shown in Figure 4.1.2.3a and Figure 4.1.2.3b. The first increments the value of `size` by seven, the second decrements it by eight.

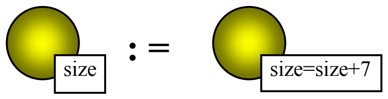


Figure 4.1.2.3a: Production 1

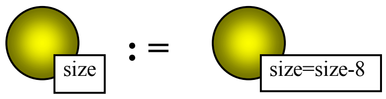


Figure 4.1.2.3b: Production 2

Suppose LevelStrategy1 is supplied with the initial graph and a single Parameter object that represents the value of the `size` variable of the single node. Thus, the ParameterEstimator object simply returns the value of `size` when asked to estimate the value for this parameter. Suppose also that the minimum acceptable value of the parameter is nineteen and the maximum is twenty.

The derivation will proceed as shown in Figure 4.1.2.3c. The strategy will pick a rule at random. If it chooses Production 1 it will apply it, see that 7 is closer to the target of 19.5 than 0, and so will keep the new graph. If it chooses Production 2, it will see that -8 is further away than 0 so it will revert to the old graph and choose Production 1, making size equal to 7.

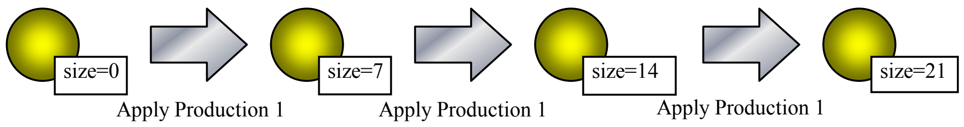


Figure 4.1.2.3c: The derivation fails to produce an acceptable graph with the rules

By the same logic, Production 1 will be applied twice more until a size of 21 is achieved. At this stage, applying Production 1 will yield a value of 28 and Production 2 will give a value of 13. Both of these are further away from 19.5 than 21 (the current value), so the strategy will terminate with failure. However, if Production 1 had been applied again, size would be 28, then if Production 2 had been applied the size would be 20, which is in the range.

The conclusion of this example is that sometimes it is necessary to apply a rule that makes no progress towards achieving the desired value of a parameter before finally a value in the range can be achieved. This is a common problem with hill climbing, and is caused by the fact that only the states that can be reached through one rule application from the current state are examined.

In the example above, a size of 21 is known as a *local maximum*, because it is not an acceptable value in itself and all rules produce a value further away from that desired. In other cases a value could be a *plateau*, which would mean that all rules produced a graph with exactly the same value of the parameter as the current position.

The solution to this problem, which has been implemented in LevelStrategy1, is to apply a production as a last resort if none of them yield an improvement. In the above example, if Production 1 had been applied when size was 21, the resulting graph would have a size of 28. Then, the system would see that Production 2 resulted in a size value of 20, which is nearer to 19.5 than 28, so this rule would be applied, resulting in success.

However, it may be the case for some problems that it is impossible to derive a graph that has a parameter of the required value. To avoid the system going into an infinite loop, the number of times that a production can be applied as a last resort is restricted, so that if this number of times is reached, the system will exit with failure.

4.1.3 Example

The LevelStrategy1 class was implemented with little trouble, and proved in tests to be successfully working. In Figure 4.1.3a is an example showing the system successfully generating a graph representing a level from a small set of rules. The rule StartProduction turns the initial Level node into a Start node, three Room nodes and an End node, where Room nodes represent rooms in the level, and Start and End nodes represent the start and end of the level respectively. AddRoom1Production and AddRoom2Production add a Room node to the level in different ways, and RemoveNode1Production removes a Room node from the level, but only where the node to be removed is not connected to other nodes that would be left unconnected to the rest of the graph if the rule was applied.

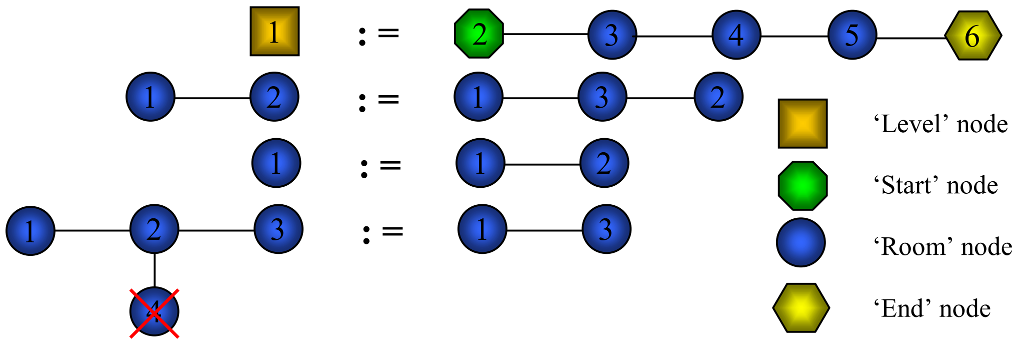


Figure 4.1.3a: The productions in the system. From the top - StartProduction, AddRoom1Production, AddRoom2Production, RemoveNode1Production

An example derivation using the above rules and starting with an initial graph of one Level node is shown in Figure 4.1.3b. Only one parameter was input to LevelStrategy1, that being the size of the level required. In this example, a ParameterEstimator class was created that measures the size of a level as the sum of the size of each of the nodes, where Room nodes have a size of two and all other nodes have a size of zero. The target size was between fifteen and thirty.

Firstly, StartProduction is applied resulting in a size of six. This is too small, so AddRoom1Production and AddRoom2Production were applied continuously until the size was finally sixteen, which is in the range required.

4.2 Estimating Parameters

In the previous section, the use of ParameterEstimator objects was discussed as a way of estimating the value of a given parameter, such as size, for a particular graph representing a level. The system has been designed so that users can create and use their own game-specific ParameterEstimator objects, but the Dungeon Generation System will come with a selection of ParameterEstimator objects, and it is these that will be described in this section.

A major constraint on these ParameterEstimator objects is that because they should be able to be used to assess a level for any game, they have to be general-purpose and cannot assume too much about the layout or objects in a game. For example, an estimation for difficulty could be done by simulating playing the level and seeing how the player's health varies during the game. However, this requires knowledge of how health is measured in the game and how objects affect it, which could be different for each game. The need for generality will limit the effectiveness of any estimation method for a particular game, but the example game rule set that will be developed at a later stage will have specific ParameterEstimator objects created for it that can use more specific and sophisticated estimation algorithms.

This section is split into three parts, one for each of the three parameters size, difficulty and fun-value. The factors in a game that affect each parameter will be discussed before details of the actual ParameterEstimator objects created to estimate that parameter are presented.

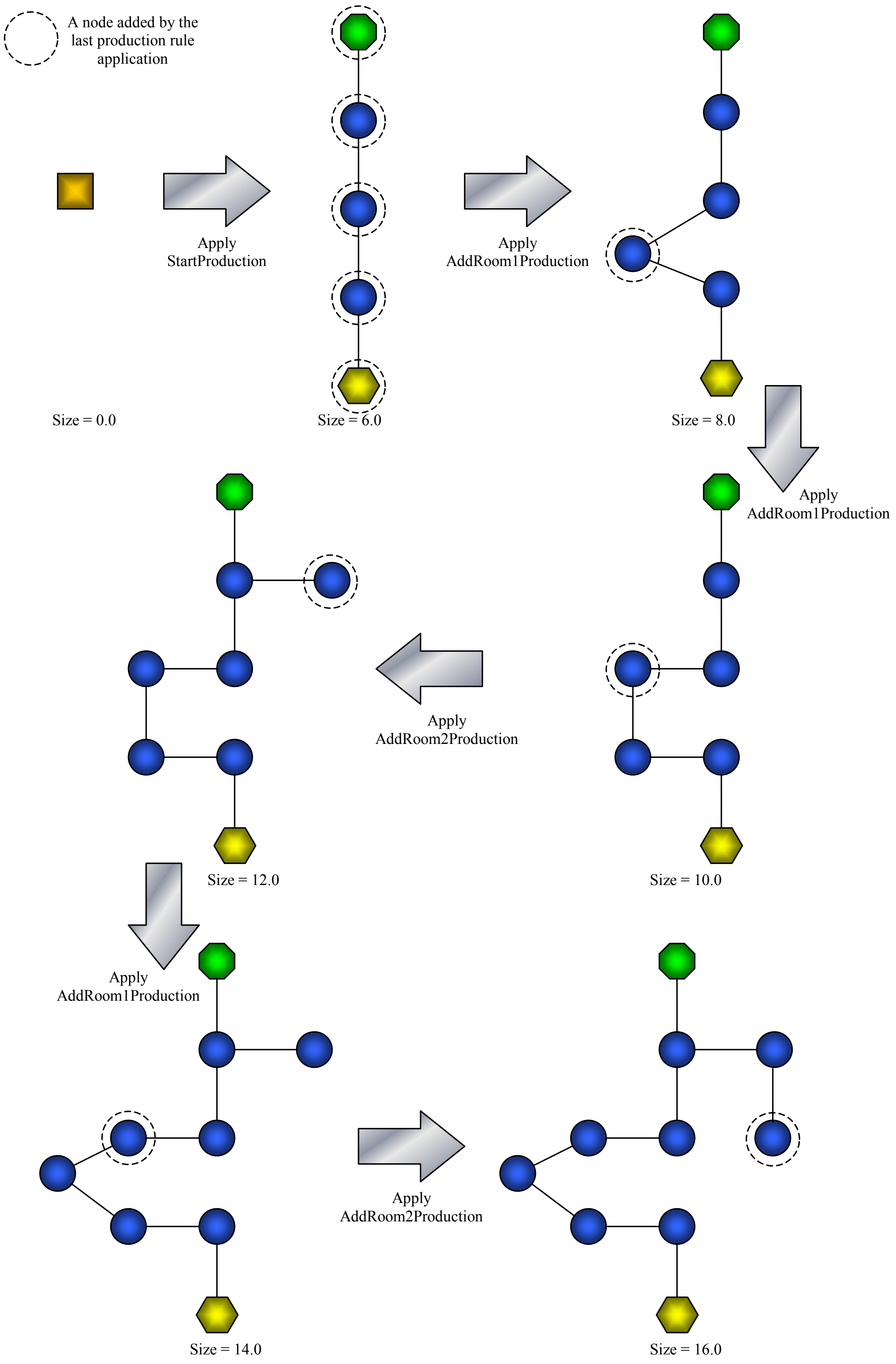


Figure 4.1.3b: An example derivation using LevelStrategy1

4.2.1 Size

4.2.1.1 Factors Affecting Size

Based on the results of the research into level design (Section 2.5), it is felt that the following measures are either ways of measuring the size of a level, or are factors in determining the size of a level: -

1. The physical size of the level (its geometric area).
2. Average distance travelled to complete the level.
3. Number of objects in the level (such as opponents, doors, bonuses).
4. Average time taken to complete the level.

Measuring the geometric area of a level will be impossible, due to the fact that the strategy will only generate a topological description of a level. Method 3 has certain disadvantages, due to the fact that on average players do not visit all rooms in a level and therefore do not encounter all of the objects. Methods 2 and 4 are similar in that they try to take into account that a player might not explore the entire level by returning an average value, which should turn out to be more realistic.

It is thought that out of these two, measuring the average time taken would be the more realistic estimate of size. This is because without geometric information, distance cannot be measured directly. Also, when one talks about the size of a level, one is really referring to how long it will take the player to complete it. This is not just a measurement of distances to be travelled, but takes into account the contents of each room, because clearly it will take less time to navigate an empty room than one full of opponents and puzzles.

4.2.1.2 Size Estimation Algorithms

Each of the four ParameterEstimator objects that have been created to estimate the size of a level will now be described.

SizeEstimator1

This is the simplest of the ParameterEstimator classes, and measures the size of a level as the number of nodes in the level. This requires absolutely no assumptions to be made about games, or indeed levels, but it is obviously not a good indicator. Some types of nodes will affect the size of a level more than others and some, such as a node to mark the start of a level, do not actually add to the size at all, since they just act as a marker rather than being an area the player must explore or an object they will encounter. Also, since the underlying graph grammar system being used allows attributes to be associated with nodes, the size of a level may depend greatly on the values of these attributes, which are not taken into consideration by merely counting the nodes.

SizeEstimator2

SizeEstimator2 measures the size of a level as the sum of the size of the nodes in the level. To do this, a special node type called AreaNode is provided in the dgs package. The AreaNode class is designed to represent a physical space in the level that objects may inhabit, such as a room, corridor, arena or staircase. Objects inhabiting that area are specified as attributes of the class. To be used, the AreaNode class must be extended, and an implementation must be provided for each of the three abstract methods getSize(), getDifficulty() and getFunValue().

Therefore to calculate the size of all components, SizeEstimator2 iterates through the list of nodes. If the current node is a subclass of AreaNode, it calls the object's getSize() method and adds the value returned to the total size so far. An example of this estimator working was shown in Figure 4.1.3b. Here RoomNode extended AreaNode, and because each room held no objects, the getSize() method just returned 2.0 for each RoomNode.

The advantage of this estimation algorithm is that by assuming very little it can provide a more accurate assessment by delegating much of the estimation to the getSize() method of node classes that the user has implemented. These user-written methods can of course assess the size of the node in a game-specific way, while SizeEstimator2 itself can be used with any game. The only assumption it makes is that a level consists of physical areas, which is true for all dungeon games.

SizeEstimator3

Like many of the provided ParameterEstimator objects, SizeEstimator3 is a combined approach, in that it assesses the size of each node as SizeEstimator2 does, but contains an additional factor of the connectivity of the graph. The idea is that when the size of a level is viewed as the time it takes to complete the level, then the more choices the user has at a particular node, the longer it will take them to complete the level.

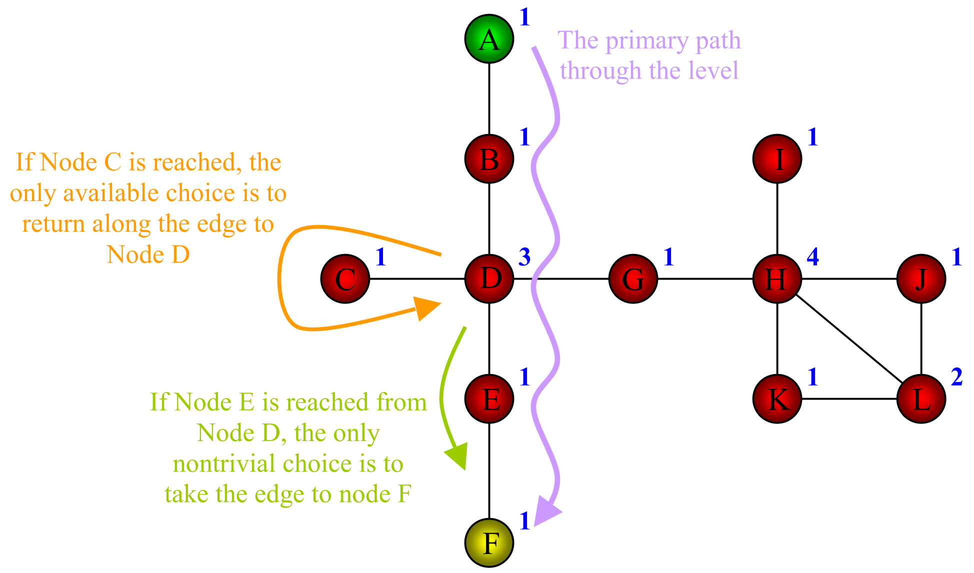
In the case of graphs, the number of choices for each node is the number of edges leaving that node. This is defined in the algorithm as: -

```
    N = Number of edges leaving node
    If N == 1 then N else N-1
```

The above algorithm will now be explained. If N is the number of edges leaving a node, then the real number of choices will be one less than this number, because the player will have to have come down one of the edges to reach that node. The second line specifies that 1 should only be deducted if N was originally greater than 1.

This is because 1 would represent a 'dead-end' node, such as Node C in Figure 4.2.1.2a. It can be argued that the number of choices here is the same as for a node with two edges, such as Node E in the example. For the dead-end node the only choice is to return along the edge that you reached the node by, and for a node with two edges the only nontrivial choice is to take the edge that you did not reach the node by. Calculating connectivity in this fashion produces better results.

With an algorithm to assess the connectivity of one node, the connectivity of the graph as a whole is simply the average of the connectivity of each of its nodes. For example, the connectivity of the graph in Figure 4.2.1.2a is 1.5. This is normalised to a value between 0 and 1 and then scaled by 10 before being added to the size factor achieved by SizeEstimator2.


The primary path through the level
If Node C is reached, the only available choice is to return along the edge to Node D
If Node E is reached from Node D, the only nontrivial choice is to take the edge to node F
A
1
B
1
C
1
D
3
E
1
F
1
G
1
H
4
I
1
J
1
K
1
L
2

$$\text{Total connectivity} = \frac{1+1+1+3+1+1+1+4+1+1+1+2}{12} = \frac{18}{12} = 1.5$$

Figure 4.2.1.2a: The connectivity of an example graph

SizeEstimator4

The problem with estimating the size of a level based on the size of all of its components is that most players do not visit all of the areas of a level when playing a game. Some rooms may only contain objects that are not needed to complete the level, such as rewards or opponents. SizeEstimator4 deals with this problem by looking at the areas on the *primary path*.

The primary path is the shortest route through a level from the Start node to the End node, in terms of the fewest intermediate nodes visited. For example, in Figure 4.2.1.2a above, if A was the Start node and F was the End node, the primary path would be [A, B, D, E, F]. An explanation for how the system calculates the primary path through a level is given in the Section 4.3. SizeEstimator4 calculates the sum of the size of each AreaNode on the primary path and averages it with the size of all AreaNodes to try to get an idea of the size of the level experienced on average by the player. The averaged value is added to the complexity value, which is calculated in the same way as for SizeEstimator3.

4.2.2 Difficulty

4.2.2.1 Factors Affecting Difficulty

The following factors have been identified as affecting the difficulty of a level: -

1. The number, type and concentration of opponents.
2. The number of health packs.
3. The number and difficulty of puzzles.
4. The relative positions of health packs and opponents.
5. The precise placement of opponents (opponents hiding behind things are more difficult to deal with).
6. Doors and keys.
7. Having to perform difficult manoeuvres (such as jumping across a wide gap).
8. Time pressures.
9. Level layout (lots of alternative paths can make things confusing).
10. The weapons and ammunition a user has (or can get) at any point in a level.

It is clear from the list above that there is no one measure of difficulty, just a number of factors that contribute to it. Some of these, such as 5 and 7 cannot be measured at a topological level. Most rely on the number and distribution of certain objects that either increase the difficulty of a level (such as opponents) or decrease it (such as health packs). The main challenge facing difficulty estimation algorithms is therefore to find a general way of estimating the difficulty of a level without making reference to any particular kind of object, such as opponents, which cannot be assumed to appear in a particular game.

4.2.2.2 Difficulty Estimation Algorithms

The ParameterEstimator objects that are designed to estimate the difficulty of a level generally work in a similar way to those that estimate the size of a level, so they will only be described briefly.

DifficultyEstimator1

The algorithm for this ParameterEstimator works the same way as SizeEstimator2, in that it estimates the difficulty of a level by the difficulty of all the nodes in the level. The only difference is that it calls the getDifficulty() method of each AreaNode instead of the getSize() method.

DifficultyEstimator2

This ParameterEstimator works like DifficultyEstimator1, except that it includes a factor measuring the connectivity of the graph representing the level as well. It is thought that the higher the connectivity of the graph, the more choices there are for the player, so the more likely they are to get disoriented and for the level to become more difficult.

DifficultyEstimator3

DifficultyEstimator3 estimates difficulty in a similar fashion to the way that SizeEstimator4 works, in that it takes into account the difficulty of AreaNodes that are just on the primary path.

4.2.3 Fun-value

4.2.3.1 Factors Affecting Fun-Value

The following factors have been identified as affecting the fun-value of a level: -

1. How difficult the level is (it should not be too easy or hard).
2. How different it is from other levels in the game.
3. Variation/originality (the variation in the level in terms of the structure of certain sections and object placement).
4. The number and type of interesting objects (such as opponents).
5. The weapons available to use.
6. The level of suspense (such as monsters jumping out from somewhere).
7. How the level looks (such as the textures used on walls).
8. The rewards (bonuses) available.
9. Resource balance.
10. The level of reinforcers.

Again, some of the factors listed above require information that is not available at the topological stage of level development, such as the available weapons, the level of suspense and how the level looks. Also, since the Dungeon Generation System generates levels independently of each other, it cannot take into account how different a level is from those already in the game. What can be abstracted out is that certain objects increase the fun of the game more than others, and any algorithm should accommodate this.

The last two factors in the list are very interesting. As explained in Section 2.5.3 and by [5], resource balance concerns the careful positioning of opponents and rewards so that a player is constantly close to losing the game but rarely does. If this is achieved then the game becomes much more exciting to play.

In Section 2.5.2, it was stated that reinforcers are rewards the player receives for performing a particular action. If the player expects a reinforcer at a particular point and one is not provided (or a smaller reinforcer than expected is provided) then the player may feel frustrated, which will obviously reduce their enjoyment of the level. These last two factors require a high degree of game-specific knowledge to be assessed usefully, and therefore will not feature in any of the generic algorithms described below, but they will be borne in mind when creating game-specific ParameterEstimator objects.

4.2.3.2 Fun-Value Estimation Algorithms

There are two ParameterEstimator objects provided to estimate the fun-value of a level. These are described briefly below.

FunEstimator1
Estimates the fun-value of a level as the fun-value of all the AreaNodes in a level, by calling the getFunValue() method of each AreaNode in the graph representing the level.

FunEstimator2
Takes into account the nodes on the primary path when estimating the fun-value of a level, in a similar way to SizeEstimator4.

4.3 Creating Good Levels

As mentioned in Section 2.2.4, one of the main drawbacks of the level generation system created by Simon Ince is that levels were often generated that had *pointless areas*. This section will detail the design and subsequent implementation of a second strategy for generating levels, which aims to make levels better by eliminating pointless areas in them.

4.3.1 The Pointless Area Problem

Pointless areas will be defined here as areas of a level that the player neither has to visit in order to complete the level, nor that reward the player sufficiently for them to choose to visit there.

Consider the level in Figure 4.3.1a. The player can complete the level by just visiting the set of nodes [1, 2, 3, 4, 8], which is the *primary path*. However, there is another way of navigating the level, and this is by the path [1, 2, 5, 6, 4, 8]. These are the only two sensible paths through the level, because any others would involve travelling down dead-ends, and therefore could be simplified to one of the routes mentioned. These sensible routes will be called the set of *paths* through the level, and any node that appears on one or more paths will be known as a *useful node*.

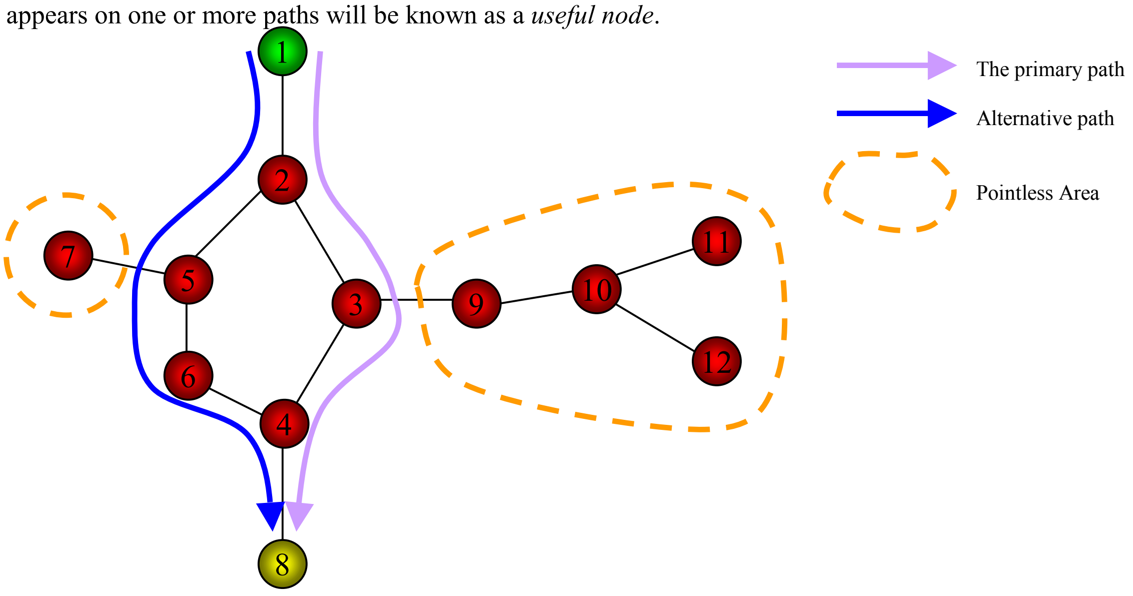


Figure 4.3.1a: An example level showing paths and pointless areas

The set of *pointless nodes* will be defined as all the nodes n such that n does not appear on a path through the level. Using this definition, a *pointless area* A can be defined as a set of nodes such that: -

1. Each node in the set is a pointless node.
2. For each node n in the set, all other nodes in the set can be reached from n without passing through a useful node.
3. There does not exist a pointless node n_1, such that $n_1 \notin A$ and n_1 can be reached from a node in A without passing through a useful node.

For the example level, there are two pointless areas. These are the sets {7} and {9, 10, 11, 12}.

So how can levels be generated so that they do not contain pointless areas? There are three possible solutions to this problem: -

1. Design the rules in such a way that pointless areas cannot occur.
2. Make the number of pointless areas in a level a parameter, in the same way that size and difficulty are parameters.
3. Create a new level strategy that deals with the problem in the strategy rather than in the rules, by altering the path that the player must take to remove pointless areas.

It will now be argued that 1 and 2 are not realistic options. It would be very difficult to design the rules so that levels generated remained sufficiently random and interesting but yet did not contain pointless areas, and also the content of the rules is out of the Dungeon Generation System's control. Option 2 is easy to implement, but would be very inefficient. Removing pointless areas from a level is a complex task, and it is therefore unlikely that a single graph transformation could accomplish this. Therefore, it was decided that Option 3 was the best choice.

4.3.2 Design

4.3.2.1 General Design

The new strategy that will be created will be called LevelStrategy2. As was mentioned in Section 2.2.1, the way in which most level designers create a new level is by first creating a map showing the rooms and corridors in the level, and then populating the level with objects. This approach will be adopted by LevelStrategy2. At the most general level, it will therefore perform the following steps: -

1. Generate a level containing no objects, only areas such as rooms and corridors.
2. Modify the level to remove pointless areas.
3. Populate the level with the remaining objects.

For 1 to be accomplished, the system needs to split up the production rules into two categories - those that modify the physical structure of a level and those that modify the contents of a level. These will be known as *room productions* and *object productions* respectively.

Firstly, the strategy should apply the room productions to an initial graph. To make sure the level is of a suitable physical size, it should take as input a separate size parameter that should specify the size required before objects have been placed in the level.

The strategy firstly finds the primary path p through the level using the breadth first search technique ([12] and [34]). This algorithm uses similar ideas to the famous algorithm by Dijkstra ([12] and [34]) that can be used to find the shortest path between two states.

The algorithm is called breadth-first search because the node it explores next is always the one that is the highest up the search tree that has not yet been explored. Due to this, it will always find the shortest path in terms of the number of nodes between the start node and a goal node.

It works by having two lists called `Open` and `Closed` that store search nodes. A search node s contains a state, but it also has a `parent` field that points to the search node from which s was discovered. `Open` stores all those nodes that have yet to be expanded (explored) and `Closed` stores all of those nodes that have been expanded. The algorithm then proceeds as follows with initial state s (adapted from [34]): -

```
Set Open equal to $\{s\}$.
WHILE Open is not empty AND $c$ is not a goal node
    Let $c$ be the front node in Open. Remove $c$ from Open.
```

```
        Find the successors $S$ of $c$.
        FOR each $n \in S$
                IF $n \notin$ Open AND $n \notin$ Closed THEN
                        Add $n$ at the back of Open.
                        Set the parent field of $n$ to be $c$.
                END IF
        END FOR
        Add $c$ to Closed.
END WHILE
IF $c$ is a goal node THEN
        Recover the path by following the trail of parent pointers back to the start node $s$.
END IF
```

After finding the primary path p through the level, the system then tries to find all other paths. For all nodes $n \in p$ it tries to find another path that goes from n to the end node without travelling along the edge from n to the next node in p. It finds all such different paths. Then it calculates the pointless areas. It does this by firstly constructing a list of all nodes that do not appear on any path. Then, it finds the pointless area that each pointless node n is in by performing a search setting n as the start node (the end node remains unchanged), with the constraint that no edge can be traversed that leads to a node that appears on a path. Since the end node must be on a path, the search is guaranteed to fail, but on failure the `Closed` list (see above) will contain precisely those nodes in the pointless area.

The strategy will have three choices as to how to deal with each pointless area: -

1. Do nothing - if the pointless area consists of a single node then on some occasions it may be left as it is.
2. Place one or more rewards in the pointless area.
3. Place a key (or similar object) in the pointless area, with a related door (or equivalent object) placed in a position so that the level cannot be completed without the key being retrieved.

It should be noted that the player is not forced to pick up rewards, but if they are significant enough to motivate the player to visit the pointless area, then the same result will have been achieved. In the language of game theory (see Section 2.5.1), the payoff to the player of exploring these areas must be greater than the payoff of avoiding them. Therefore, the set of rules applied to the level once rewards have been placed in pointless areas must make sure that the loss to the player of reaching the rewards (such as decreased health from encountering opponents) must be less than the rewards themselves.

The final stage is to apply the object productions to the graph that has been generated so far, and this will yield the final level.

4.3.2.2 Class Design

The main classes in the DGS that concern LevelStrategy2 are listed in Table 4.3.2.2a.

Name	Features	Purpose
Activator	extends LevelObject abstract	A general object to represent an item such as a key or switch.
AreaCreator	extends PartialStrategy	Produces a graph just containing areas and no objects from an initial graph.
DGSEdge	extends Edge	An edge that allows Responders to be placed on it.
EndNode	extends Node	A node used to mark the end of a level.
LevelObject	abstract	Represents an object in a level that can be stored at a node.
LevelPopulator	extends PartialStrategy	Produces the finished graph by applying object productions to a graph that has had all pointless areas dealt with.
LevelStrategy2	extends LevelStrategy	The new level strategy, designed to remove pointless areas from levels.
PartialStrategy	abstract	A substrategy that performs a stage of development on a graph, such as adding rooms, rather than generating an entire level.

Path		Represents a path through a level.
PointlessArea		Stores data on a pointless area, such as the nodes in it.
Responder	extends LevelObject abstract	A general object to represent an item such as a door; an object that cannot be passed until one or more conditions are satisfied.
Reward	extends LevelObject abstract	Represents a reward, such as a health pack.
Search		Contains methods to search a level in order to find paths through it.
StartNode	extends Node	A node used to mark the beginning of a level.

Table 4.3.2.2a: The main classes in the dgs package concerning LevelStrategy2

Many of the objects in the table above will be explained in much greater detail in a later part of this section. It is worth saying that AreaCreator and LevelPopulator are just subcontractor objects that perform Stages 1 and 3 and just exist to reduce the length of LevelStrategy2.

As stated in the introduction to this section, the system needs to be able to calculate paths through the level in order to find and deal with pointless areas. For this to be possible some constraints must be placed on the content of graphs. There must always be a single StartNode and EndNode in the graph, marking the beginning and end of the level. Also, all edges must be DGSEdges and not Edges to allow objects such as doors to be placed on edges. The only other constraint is that there must only be one node connected to the StartNode and one node connected to the EndNode.

4.3.2.3 Placing Rewards

Rewards are items, such as health packs and ammunition, which are useful to the player, but not strictly necessary in order to complete a level. They are most likely to be used as the solution to small and medium-sized pointless areas (up to five nodes in size). The Reward class was introduced so that the strategy could be applied to any game. Level objects that are rewards are declared in their own class that extends the Reward class; HealthPack, Ammunition and Weapon are some possible examples. One of the arguments passed to LevelStrategy2 is a set of Reward objects that it can use. If no Reward objects are passed to it, it will not try to deal with a pointless area through rewards.

The algorithm for placing rewards in pointless areas proceeds as follows: -

1. Calculate the target reward size - each type of reward has a size of either small, medium or large. If the pointless area is large, the system will try to put a bigger reward there than if it is just a single node. As with the entire algorithm, there is some random element in deciding on the target reward size, and the system checks first that there is a reward type with the size it wants.
2. Choose a reward type r based on the target reward size.
3. Find the furthest away node in the pointless area (in Figure 4.3.1a these would be 7 and 11/12 respectively) and put an instance of r there.
4. Choose a random (but proportional to the size of the pointless area) number of other rewards and place these randomly in the nodes of the pointless area. This is for larger pointless areas, where placing a single reward would probably not be a large enough reinforcer to cause the player to enter the area.

4.3.2.4 Placing Activators

In order for LevelStrategy2 to be game-independent, it could not be assumed that a game contained keys and doors, for instance. More general structures needed creating, and these have been called *activators* and *responders*. Activators are objects that are activated by a player in some way. A key is an activator that is activated by picking it up, and a switch is an activator that is activated by pressing it.

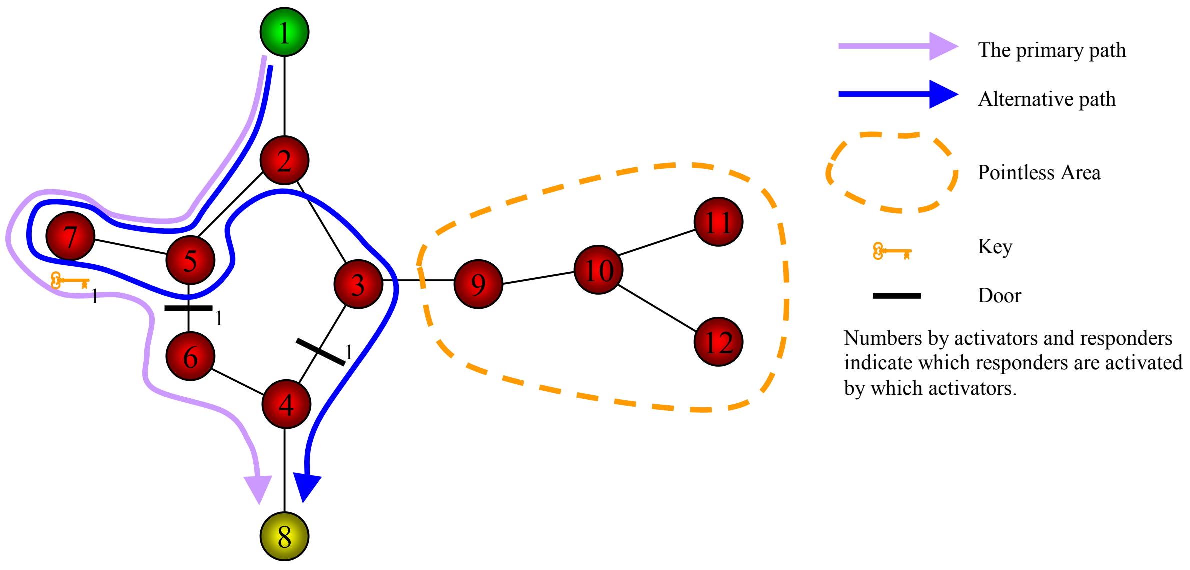
Responders are barriers of some sort that are placed on edges. The player cannot traverse the edge unless the activators that are related to that responder have all been activated. So for a door that needs a key, the door cannot be opened (and passed through) until the key that matches it has been picked up. If a responder was placed on the edge leading to the end node, it could also indicate a mission objective of some sort, as is common in some dungeon games.

As with rewards, specific types of activators and responders extend the Activator or Responder class. So Key could extend Activator and Door could extend Responder. Each Responder has a list of compatible activators associated with it, which indicates which Activators can be paired with the Responder. For example, keys commonly open doors, so Key would be a compatible activator of Door.

The algorithm for placing activators and responders is complicated, due to the care that needs taking in order to make sure that the level remains possible to complete after responders have been placed. It is described below: -

1. Pick a responder type r at random.
2. Pick an activator type a at random, where a must be compatible with r.
3. Find the furthest away node in the pointless area and put an instance of a there.
4. Some responders (for example, those representing mission objectives) may only be able to be placed at the end of a level. If r is such a responder, place an instance of it on the edge leading from the end node and finish.
5. There will be one and only one useful node n that a node in the current pointless area will be connected to. Choose an edge on the primary path at random, such that if n appears on the primary path then this edge occurs on part of the path after node n. For example, consider the pointless area {9, 10, 11, 12} in Figure 4.3.1a. Here n is Node 3, and the chosen edge can either be that connecting Nodes 3 and 4, or the edge connecting Nodes 4 and 8, because these two edges are after Node 3.
6. Perform a search to see if the level can still be completed without activating the activator. This could occur if there were multiple paths through the level and the responder was placed on an edge that did not feature in all paths.
7. If the responder can be bypassed then something needs to be done about it. Based on a random factor, do either of two things: -
    a. Remove r and place it on an edge past the point where all paths have converged. In Figure 4.3.1a, there is only one edge after both paths have converged, and this is the edge between Nodes 4 and 8.
    b. If the type of responder allows multiple responders to be activated by the same activator, place more responders on other paths until all are blocked.

After the pointless area has been dealt with, the activator will cause the paths through the level to have changed. For example, if an activator was placed in Node 7 and responders were placed on the edges Node5-Node6 and Node3-Node4, the paths would now be as shown in Figure 4.3.2.4a. Therefore the pointless areas need recalculating. Also, whenever a pointless area has been dealt with by placing activators and responders, the pointless areas should be found again, because there still may be some nodes in the old pointless area that still do not appear on any path. For example, if an activator was placed in Node 11, Nodes 9, 10 and 11 would now need to be visited to complete the level, but Node 12 would not, and so would still be pointless.


1
2
3
4
5
6
7
8
9
10
11
12
1
1
1
1
The primary path
Alternative path
Pointless Area
Key
Door
Numbers by activators and responders indicate which responders are activated by which activators.
Figure 4.3.2.4a: An example showing how the paths in a level change

Before moving on, it is also worth mentioning the difficulties encountered in adapting the search algorithm to allow for activators and responders. These changes needed to be made because the Dynamic Programming Principle (see Section 4.1.1) breaks down for levels with activators and responders. This is because the problem of finding a path through a level with activators and responders no longer has the optimal substructure property. For the level in Figure 4.3.2.4b, the optimal solution to get from Nodes 2 to 4 is no longer a subproblem of getting from Nodes 1 to 5, as it would be if the level contained no activators and responders.

The Dynamic Programming Principle is used in breadth-first search to enable each search node to only have to store one node that it could be reached from (using the `parent` field). This is no longer the case, because now some nodes will need to be visited more than once (such as Node 5 in Figure 4.3.2.4a).

It will also mean that the parent pointer can no longer be used to recover path information. If a node is visited more than once, it will only store a pointer to the last node it was reached from, thereby losing all information on how it was reached on previous occasions. For the example above this would cause the reverse path to be calculated as [8, 4, 6, 5, 7, 5, 7, 5, 7 ....] because 5 points to 7, even though the first time 5 was visited it was actually reached from 2. Therefore it would go into an infinite loop.

Nothing was found in the literature on search algorithms that provided a answer to this problem, so a new one was developed. The solution eventually found involves tricking the search algorithm into thinking that when a node is visited more than once it is actually a different node being visited. Whenever an activator is activated, all nodes are cloned. An edge is formed between the node the activator was at and the cloned version of this node. The list of activators that had been activated up to that point is also copied, and the newly activated activator is added to the list. A morphism (relation) is kept between each cloned node and its original node, so that all cloned nodes in the final path can be converted back to the original nodes at the end. An edge can only be traversed if for each responder on the edge, all of its related activators have been activated. The goal node has to be altered to become not only the End node, but all clones of the End node as well.

A diagram showing how this would work for a simple level is shown in Figure 4.3.2.4b. The strategy will produce the path [1, 2, 3, 6, 6a, 3a, 4a, 5a]. It will then replace all cloned nodes with their original node to get [1, 2, 3, 6, 6, 3, 4, 5]. Finally it will merge identical nodes that appear consecutively in the path, to finally achieve the correct answer of [1, 2, 3, 6, 3, 4, 5].

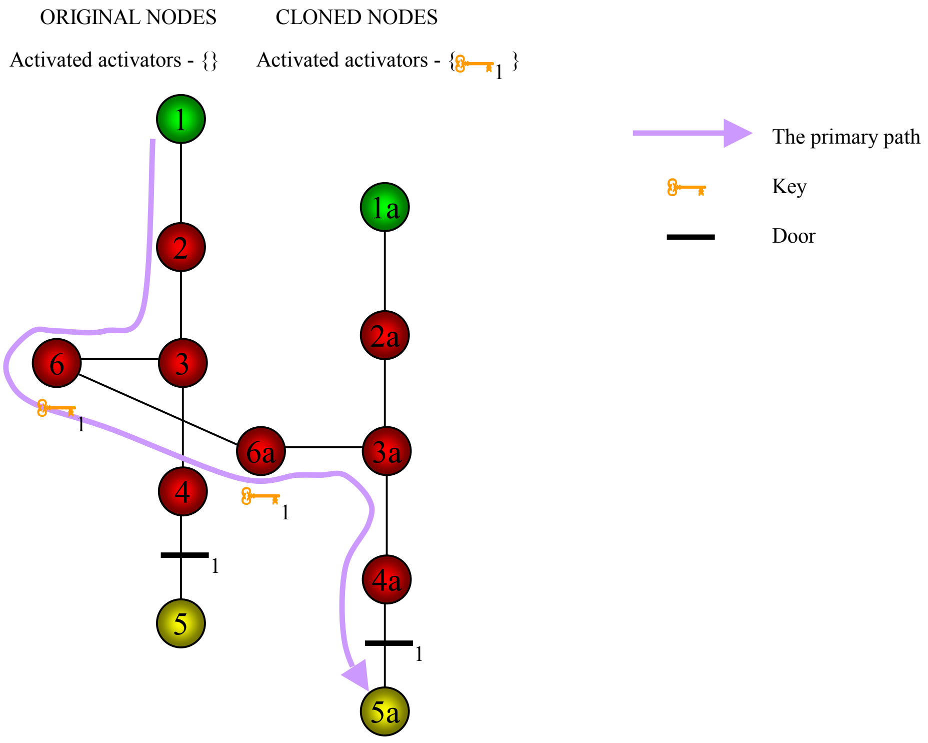


Figure 4.3.2.4b: An example showing how the search algorithm works with activators and responders

4.3.3 Example

The following example in Figure 4.3.3a shows LevelStrategy2 working. The set of available responders to use was {Door}. The set of available activators was {Key, Switch} and {HealthPack, Ammunition, Weapon} could be used as rewards. The graph after the rooms had been created was the same as that in

Figure 4.3.1a. The set of object productions consisted of two productions to add and remove opponents from a room.

LevelStrategy2 works very well and places rewards and activator/responder pairs intelligently so as to eliminate pointless areas. The only problem with the algorithm is its efficiency. Depending on the number of activators placed in the level and the positions of the corresponding responders, the strategy can sometimes take an unacceptable amount of time to produce the level.

One of the main causes of this slowness lies in the algorithm to find all of the paths through a level. To do this, a search is required for each node off each path found so far, and when there are many activators not only are the number of paths increased, but also each path contains many more nodes.

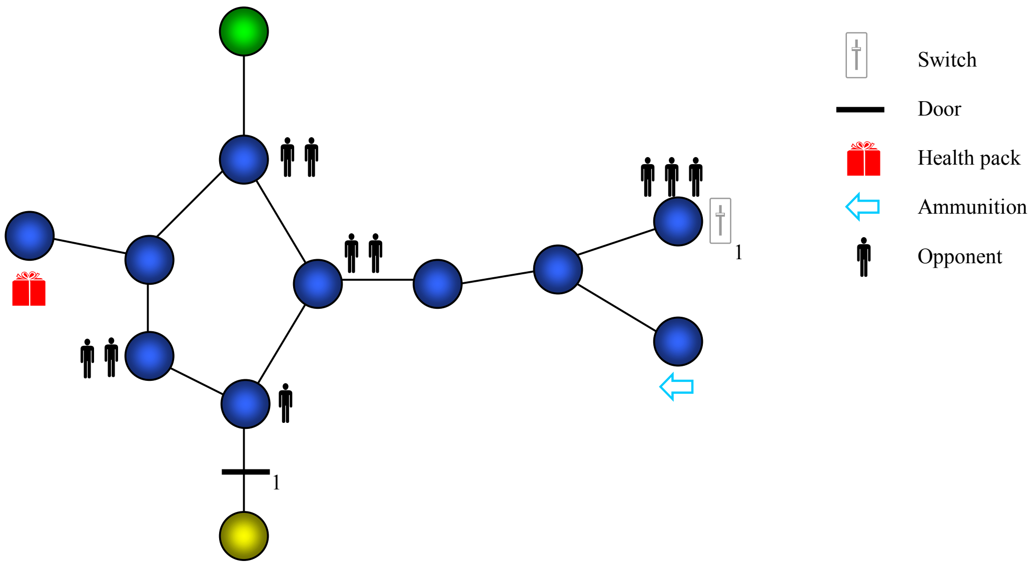


Figure 4.3.3a: An example of a level generated by LevelStrategy2

4.4 A Graphical User Interface

4.4.1 Introduction

Up until this point, to use the Dungeon Generation System a person must create their own Java file giving details of all the parameters to be passed to the LevelStrategy they wish to use, such as what productions to use, the start graph and the Parameter and ParameterEstimator objects to be used. This has several disadvantages. Firstly, it means that the system can only be used by those proficient in Java and secondly the files must be recompiled every time the user wishes to generate a level with different characteristics.

Therefore it was decided to create a graphical user interface (GUI) for the system to improve its usability. In an extension to the project, it was also decided to make this program be able to graphically display graphs representing levels, thereby helping to visualise results and providing the first step towards creating geometric descriptions of automatically generated levels.

4.4.2 Design

After careful design, it was decided that the graphical user interface should have the following features and characteristics: -

- Allow the user to choose a strategy then be asked to input the parameters specific for that strategy.
- To allow a level to be displayed textually and graphically.
- To allow levels to be loaded and saved.

The main classes involved in the graphical user interface are listed in Table 4.4.2a.

Name	Features	Purpose
CreateLevelFrame	extends JFrame	The frame where the user selects what strategy they wish to use to generate a level.
Level		Stores the Graph object representing a level and the LevelMap object describing how the level is laid out.

LevelManager		Loads a level from file and saves a level to file.
LevelMap		Stores the layout of a particular level.
LevelMapCreator	abstract	Represents a way of laying out a level. Actual implementations extend it.
LevelStrategy1Frame	extends JFrame	Allows users to select the parameters for level generation using LevelStrategy1.
LevelStrategy2Frame	extends JFrame	Allows users to select the parameters for level generation using LevelStrategy2.
LoadLevelFrame	extends JFrame	Displays a level graphically. Allows levels to be loaded and saved.
MainMenuFrame	extends JFrame	The main menu in the graphical user interface for the Dungeon Generation System.
RandomLevelMapCreator	extends LevelMapCreator	Lays out a level in a totally random fashion for it to be displayed graphically.
SimpleLevelMapCreator	extends LevelMapCreator	Lays out a level in a way that avoids nodes and edges intersecting.

Table 4.4.2a: The main classes involved in the graphical user interface

A few features of some of the above classes will now be described. For the system to be totally flexible, the GUI had to work with any LevelStrategy, and each LevelStrategy could have different parameters. The solution to this was to allow each LevelStrategy to define its own frame where parameters were input. For the built-in strategies for the DGS, these frames have been provided.

The graph drawing section of the GUI was also designed to be flexible, with each type of node in the system defining how it is to be drawn. The system has an abstract class LevelMapCreator, which actual ways of laying out the nodes in the graph extend. Two of these are implemented in the DGS. The first, RandomLevelMapCreator, positions nodes totally randomly. This often leads to edges crossing, which may be acceptable for some graph drawing applications, but is not here (see Figures 4.4.2b to 4.4.2d). This is because edges represent doorways between areas (such as rooms). Therefore for edges to cross, two doorways would have to intersect if the rooms were to actually be placed in such positions. This is invalid because if two doorways cross it would mean that you could access the rooms leading off one doorway from the other, which is not the structure of the graph.

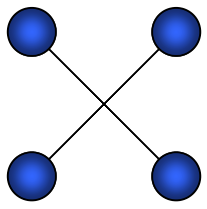


Figure 4.4.2b: A graph layout in which edges cross

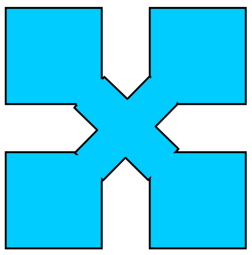


Figure 4.4.2c: The geometric interpretation of the layout

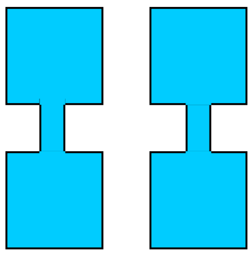


Figure 4.4.2d: A correct geometric interpretation of the textual description of the level

To combat this, a second LevelMapCreator class called SimpleLevelMapCreator was implemented. Its name refers to the fact that it produces the simplest form of geometric description of a level, merely laying out nodes and edges instead of drawing actual rooms and corridors, which is beyond the scope of this project. Provided that the graph is linear (i.e. there are no loops in it) and that no node has more than thirty-six edges attached to it, the strategy will always lay out a graph in such a way that no nodes and edges intersect. Below is a simplified description of how it works: -

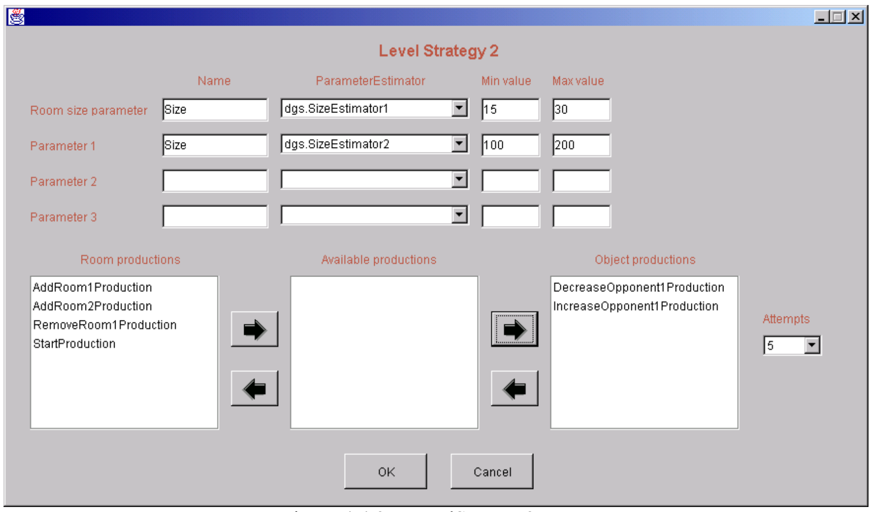
```
Place the first node at position (0,0).
WHILE there are still nodes that have not been placed
        Find a node $n_1$ connected to a different node $n_2$ such that $n_1$ has not been placed and $n_2$ has
        been placed.
        WHILE $n_1$ has not been placed
                Choose a direction $d \in \{0,10,20,30,...,350\}$ where $d$ has not already been tried
                for $n_1$.
                Set position $p$ to be distance 100 away from $n_2$ on a bearing of $d$ degrees.
```

```
                    IF $n_1$ in position $p$ does not intersect with any other placed nodes and does not
                    intersect with any edges between placed nodes THEN
                              Place $n_1$ in position $p$.
                    END IF
          END WHILE
END WHILE
```

The method performs a backtracking strategy so that if a node n cannot be placed using any of the possible directions, the last node to be placed is removed and placed in a different place, before the system attempts to place node n again. Originally, backtracking was continuously done until all nodes were successfully placed or until all possible layouts had been attempted. However, this was found to be slow (due to the exponential number of possible combinations), and it was much quicker to simply restart the layout process from scratch after a certain number of backtracks. If the process is restarted a certain number of times, then the system will give up and lay out the nodes using RandomLevelMapCreator instead.

4.4.3 Example

The graphical user interface was tested on the same set of production rules used in the example in Section 4.1.3. LevelStrategy2Frame, in which the parameters that will be used by LevelStrategy2 to generate a level are entered, is shown in Figure 4.4.3a. The top boxes on this frame are for entering the parameters used during level generation, and the bottom three list boxes are used (in order) to specify the room productions, available productions (i.e. those that have not been selected to be used in level generation) and object productions. Productions can be moved from box to box using the arrow buttons.


Level Strategy 2

	Name	ParameterEstimator	Min value	Max value
Room size parameter	Size	dgs.SizeEstimator1	15	30
Parameter 1	Size	dgs.SizeEstimator2	100	200
Parameter 2				
Parameter 3				

Room productions	Available productions	Object productions
AddRoom1Production		DecreaseOpponent1Production
AddRoom2Production		IncreaseOpponent1Production
RemoveRoom1Production		
StartProduction		

Attempts: 5

Figure 4.4.3a: LevelStrategy2Frame

Figure 4.4.3b shows the LoadLevelFrame with four different geometric representations of the same level. A new layout can be produced by choosing the ‘Redraw’ option from the ‘File’ menu. Levels can also be loaded and saved here using the ‘Load’, ‘Save’ and ‘Save As’ options, which are also available under the ‘File’ menu. In the example, node types are the same as in Figure 4.1.3a, namely circles for rooms, an octagon for the StartNode and a hexagon for the EndNode. Responders are indicated by a black line that is perpendicular to the edge the responder belongs to. Nodes with a key symbol on them indicate those at which one or more activators are present.

Clicking on a node or responder brings up a box giving details of the attributes of the node or responder, such as the opponents at a certain node, or the activators related to a certain responder. Nodes can be moved to manually alter a layout. This is achieved by firstly clicking on a node, then clicking on the new position.

4.5 Summary

The Dungeon Generation System allows the generation of levels for any dungeon game, using the Graph Grammar System as its underlying method of manipulating the graphs that represent levels. In Section 4.1, the first strategy specific to level generation, LevelStrategy1, was introduced, and it was seen that this strategy could generate levels that had certain properties, which were specified by parameters. The ParameterEstimator objects that are built into the DGS were discussed in Section 4.2. These provided different ways of measuring the size, difficulty and fun-value of a level.

Section 4.3 introduced LevelStrategy2, which was designed to create levels that did not contain pointless areas. This was achieved by putting rewards, activators and responders into levels to either encourage or force the player to visit rooms that previously there was no point in visiting. Lastly, a graphical user interface was developed for the DGS that made it easier for a user to enter the parameters needed for level generation, and also helped to visualise the results by providing a geometric description of a level by randomly drawing the underlying graph.

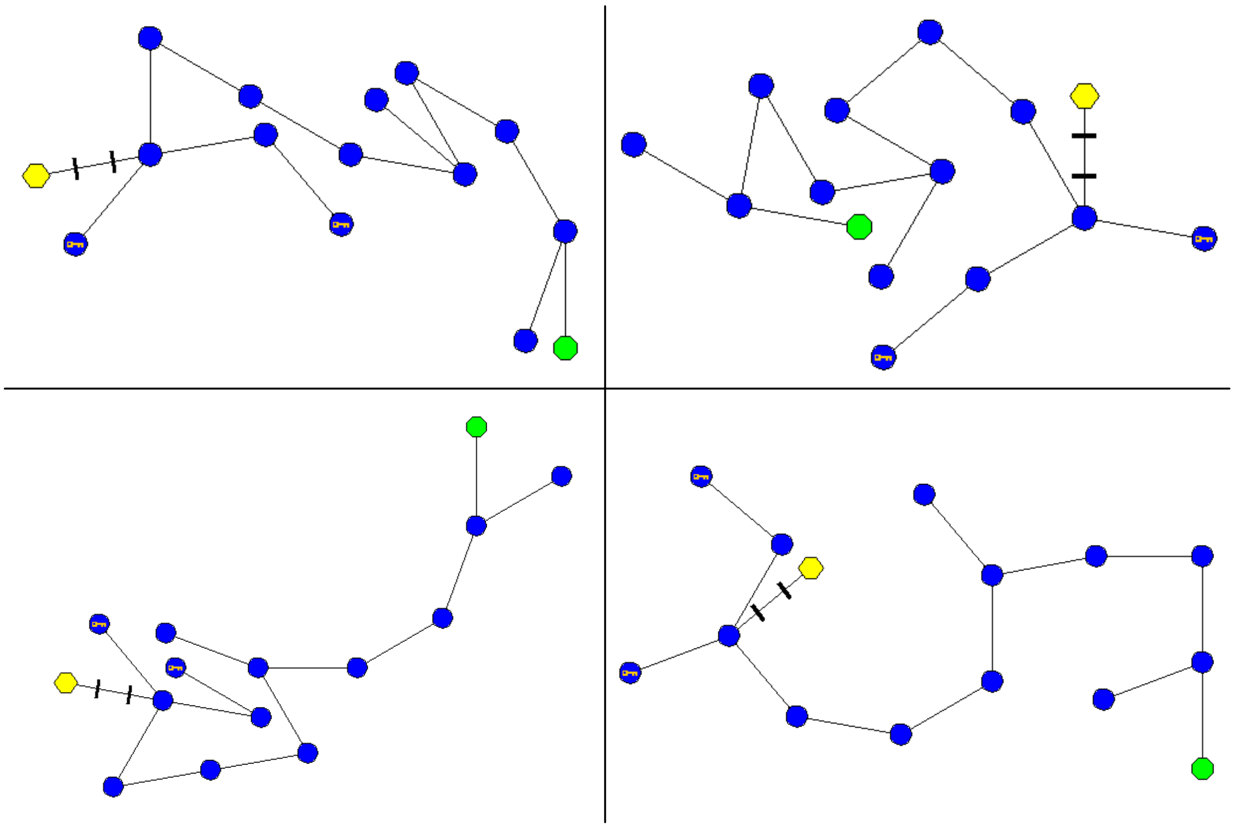


Figure 4.4.3b: Four geometric descriptions of the same topological level

Chapter 5 – Evaluation

In this chapter, the Dungeon Generation System will be evaluated. For some factors, such as quality and expressiveness, this evaluation will be achieved by creating rules for an example game and then examining the levels output. Other factors, such as efficiency, are dealt with separately. The evaluation of the system is not static, but includes improvements that have been made in light of the results of the evaluation.

5.1 Rules for an Example Game

5.1.1 Introduction

To test the capabilities of the Dungeon Generation System, it was decided to produce a set of rules and other classes to generate levels for an existing dungeon game. The game chosen for this was Dark Forces [25] due to the fact that the way the game works was well understood and it was of a suitable level of complexity to be modelled in the DGS.

Figure 5.1a: Screenshot from *Dark Forces* [25]

The game is set in the futuristic Star Wars universe and pits the player against the evil Empire. The aim of each level is to achieve one or more mission objectives. These could range from reaching a certain area of a level to retrieving an object of some kind, such as secret enemy plans.

The player has a certain amount of health, which is a value between 0 and 100 where 100 indicates the player has perfect health and 0 indicates that he or she is dead. The health of the player at the start of the game is 100. To protect the player's health there is shielding. Shielding starts off at 100 and can range between 0 and infinity (theoretically, at least). Enemy opponents do not affect the player's health as long as the player's shielding is greater than 0.

Another component of the player's state is the number of energy units he or she has. Energy units are the game's equivalent of ammunition, and are used to fire weapons at opponents. The player starts out with a few weapons and can acquire others by picking them up (sometimes off a dead opponent). All of the usual components that make up dungeon games, such as keys, doors and rewards are present. The areas the player has to explore range from rooms and corridors to more open plan areas. The player can also travel up or down using stairs, ramps or lifts.

5.1.2 Modelling the Game

5.1.2.1 Simplifications

Section 5.1.1 contained a brief summary of the game, but in fact it is quite complex. The following simplifications have been made to remove certain features that cannot be easily represented by the system and to speed up level generation: -

- Some opponents, weapons and other objects in the system have been omitted. These include the headlamp object that allows the player to see in the dark. This would have been very difficult to model in the DGS.
- Lifts have been omitted since the graphical user interface for the DGS only lays out levels in two dimensions, and therefore buildings with many floors could not be displayed properly.
- In the game, the player has lives, i.e. when they die they get restored to full health so many times before they finally lose the game. In addition, a player's health is only at its maximum at the start of the first level of the game; whatever health he or she has at the end of Level n is the health he or she has at the start of Level $n+1$. In the system, the concept of lives has been removed and it is assumed the health when starting a level will always be the maximum.
- Some simplifications were also made to simulate a player playing the game, but these are discussed in Section 5.1.2.4.

5.1.2.2 Level Objects and Node Types

The objects in the game were divided into five categories. These include the previously discussed categories of activators, responders and rewards. There are also opponents, for example ProbeDroid, that all extend the abstract class Opponent, and hazards, such as GunTurret, that are subclasses of the Hazard class. The difference between an opponent and a hazard is that the player can kill or destroy an opponent using his or her weapons, whereas a hazard cannot be killed by the player, only avoided.

All nodes in the game, apart from the StartNode and EndNode, are subclasses of SpaceNode. The SpaceNode class has four attributes to store the Opponent, Hazard, Activator and Reward objects at that node, as well as a `spaceSize` attribute. This represents the physical size of the area, which is the time required to pass through the room disregarding opponents and other objects that might slow the player's progress. It is used to estimate the size of a level.

Room, Corridor, OpenArea, Staircase and Ramp are all subclasses of SpaceNode. Each of these has a different value of the `spaceSize` attribute inherited from SpaceNode. For example, an OpenArea object, representing a larger area than a room such as an arena, has a larger value of `spaceSize` than a Room object.

5.1.2.3 Production Rules

The system contains a total of sixteen productions. These productions are similar to the ones used in the example in Section 4.1.3, except that there are extra productions to add and remove Hazard and Weapon objects from a level. There is also an extra production, AddSpace3Production, to add rooms to a level in a way that introduces nonlinearity (i.e. loops). This is shown in Figure 5.1.2.3a. This production makes levels more interesting at the expense of causing level generation to be slower and making it so that edges may cross when the level is displayed graphically (see Section 4.4.2).

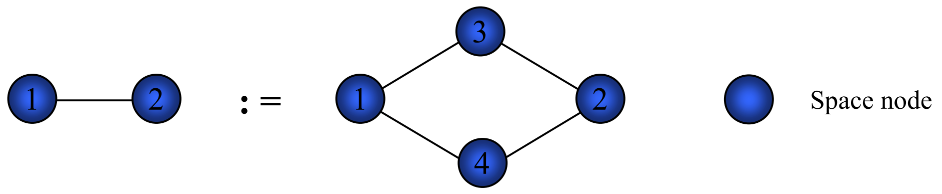


Figure 5.1.2.3a: AddSpace3Production. Nodes 3 and 4 are Corridor nodes. For the production to be applicable, Nodes 1 and 2 must be Room, Corridor, or OpenArea nodes and the type of Node 1 must be the same as the type of Node 2 (e.g. if Node 1 is a Room node then Node 2 must be a Room node)

Some room productions can only work on certain subclasses of SpaceNode. For example, ramps and staircases connect two physical areas; therefore Ramp and Staircase nodes can only be inserted between two other SpaceNode objects rather than being simply attached to a SpaceNode.

Before deciding on this set of productions, a number of variations on this rule set were tried. These included having multiple ways of expanding the graph from the initial Level nonterminal node and also having rules to describe higher-level structures such as rings of nodes. However, it was found that these did not produce more interesting levels (since any of these structures could be formed from multiple applications of simpler rules) and therefore they were not worth including because they just increased the complexity of the rule set.

5.1.2.4 ParameterEstimator Objects

Because a detailed knowledge of the game was available, a number of specialised ParameterEstimator objects have been created to estimate properties of the game's levels more accurately than is possible with any of the general ParameterEstimator objects described in Section 4.2. These new ParameterEstimator objects are described here.

Two new ParameterEstimator objects have been created – GameDifficultyEstimator and GameFunEstimator, to estimate the difficulty and fun-value of levels respectively. These both work by performing a simulation of a player playing the game and then using the information obtained from this simulation to assess difficulty or fun-value.

The simulation works by following the path of the player through the level and at each point estimating their health, shielding and energy units. These three pieces of information are referred to as the player's *state* at any particular point in time. The player's state is represented by a PlayerState object that simply records the values for health, shielding and energy, as well as recording the node the player was at and the event that triggered their state to change (such as fighting an opponent or picking up some shield units as a reward).

PlayerState objects can be written as a four-tuple (c, h, s, e) where c is the current node the player is at, h is the player's health, s is the player's shielding and e is the number of energy units they have. Using this definition, some pseudocode is given below showing how the simulation works: -

```
Find the primary path $P$ through the level.
Let $c$ be the current node the player is at. Set $c$ to the first node in $P$.
Let $S$ be the list of PlayerState objects.
Add a new PlayerState object $p = (c,100,100,100)$ to $S$.
WHILE the end of the path has not been reached
        Set $c$ to the next node in $P$.
        FOR each level object $o$ at node $c$
                Set $p_{old}$ to the last PlayerState object in $S$.
                Create a new PlayerState object $p_{new} = (c, h, s, e)$ where $h$, $s$ and $e$ are
                the values of $p_{old}$ but modified due to the encounter with object $o$.
                Add $p_{new}$ to the end of $S$.
        END FOR
END WHILE
```

The way in which a new PlayerState object is created based on an existing PlayerState object when a level object o is encountered is that each Opponent, Hazard and Bonus object in the system has an alterState() method. This takes a PlayerState object as an argument and returns a new PlayerState object reflecting the player's updated state.

The weapon the player is using is taken into account when calculating how an opponent affects a player's state. The simulation keeps track of the Weapon objects held by the player. Whenever the player encounters an opponent, the player fights the opponent with the best weapon that the player has enough energy units to use. To determine which weapon is the best, each weapon has a rating, which can be accessed by the getRating() method. A rating is a number between 1 and 10 representing the power of the weapon, with 1 being the lowest and 10 being the highest. The Weapon object that the player is using is also passed as a parameter to the alterState() method of an Opponent object, so that this can be taken into account.

In the simulation, health can fall below 0. This does not happen in the game, but is used to give a more precise estimate of the difficulty of the level. For example, if health can become negative then a final health of -1000 means the level is obviously more difficult than if the final health was -10. This information would have been lost if health was not allowed to fall negative as in both cases the health would have been recorded as 0.

GameDifficultyEstimator estimates the difficulty of a level based on the results of the simulation as follows: it iterates through the list of PlayerState objects that were created during simulation to record the state of the player at all points in the level. It assigns a difficulty to each PlayerState object, where the higher the player's health and shielding, the lower the difficulty. Difficulty is constrained to be in the range 0 to infinity.

Originally, a maximum difficulty of 300 was set, but this led to poor performance during level generation. This was because it often led to a plateau situation (see Section 4.1.2.3). For example, consider the task of generating a level with a difficulty of between 200 and 250. If a level had a true difficulty of 600, this would be constrained to 300. A rule would be applied, such as to remove an opponent, and even though the true difficulty might have fallen to 550, the value 300 would still have been returned by GameDifficultyEstimator. Thus the level strategy would have assumed that applying the rule did not yield a graph that had a difficulty closer to that desired, so the new graph would be discarded.

Once a difficulty value is calculated for every PlayerState object, the difficulty returned is calculated as the mean of the end difficulty of the level (the last PlayerState object) and the difficulty of the PlayerState object representing the worst state of the player. Difficulty is estimated in this way because if just the difficulty of the end PlayerState object was returned, this may not give a true picture. For example, the first part of a level may feature lots of opponents, leading to a situation where the player would probably die. However, if the last part of the level contained lots of rewards, such as those to boost the player's health, then the player's health by the end of the level may be quite high even though in practice the player would have died halfway through the level due to the large number of opponents. Taking into account the worst PlayerState resolves this problem.

GameFunEstimator estimates the fun-value of a level based on the percentage of time that the player's health, energy and shielding were low. This models the resource balance principle ([5] and [6]) of good level design, as discussed in Section 2.5.3, which basically says that a level is more fun to play when the player is always close to running out of important commodities, such as shielding and energy units in terms of the Dark Forces game. The fun-value of a level is also partly determined by the fun-value of objects within the level. For example, some opponents are more exciting to fight than others. To accommodate this, GameFunEstimator also uses the fun-value returned by FunEstimator2 (see Section 4.2.3.2) to estimate the fun of a level.

5.1.3 Example

Figure 5.1.3a shows an example of a Dark Forces level that was generated by the Dungeon Generation System using the nodes, level objects and productions described in Section 5.1.2. A possible two-dimensional plan of this level, which also shows the position of opponents, hazard and rewards, is displayed in Figure 5.1.3b. The parameters used in level generation were an initial size of between 30 and 40 (before pointless areas had been dealt with), a final size of between 200 and 300, a difficulty of between 75 and 150 and a fun-value of between 50 and 100. These parameter values would correspond to one of the first levels in a game, as the difficulty rating is relatively low.

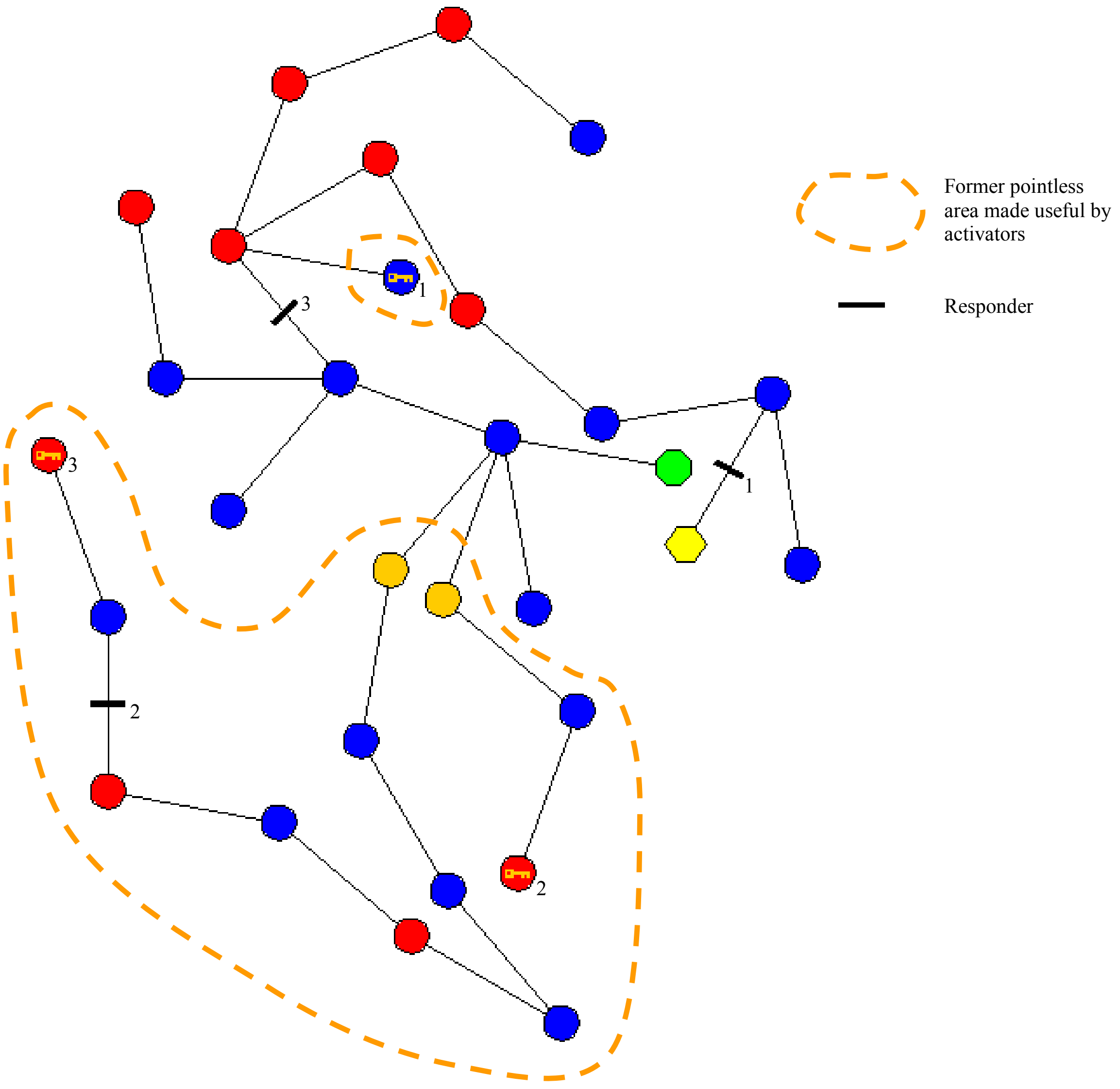


Figure 5.1.3a: A screenshot of a level for the Dark Forces game produced by the Dungeon Generation System. Numbers have been added to Activators and Responders to allow readers to see which ones match up

The two specialised ParameterEstimator objects, GameDifficultyEstimator and GameFunEstimator, were used to estimate the difficulty and fun-value of the level respectively. The final level had a size of 201.0, a difficulty of 77.5 and a fun-value of 56.0 so it fell within the range of acceptable values for all three parameters.

It can also be seen that activators have been sensibly placed to remove pointless areas, as shown on Figure 5.1.3a (other pointless areas were removed by placing rewards, which are not shown on the diagram). Furthermore, it should be noticed how intelligently these activators have been placed. None of the activators are near their related responder.

In addition, responders have also been placed to stop the player from reaching certain activators, rather than just being placed on the direct route from the Start node to the End node. An example is Responder 2, which is blocking the path to Activator 3. This shows that the level strategy takes into account that the path changes once activators are put onto it, and it realises it can place responders between nodes that used to be pointless, but are not pointless anymore.

5.1.4 Conclusions

The example rules and productions developed to create descriptions of levels for Dark Forces show that the Dungeon Generation System is expressive enough to allow it to model the workings of dungeon games. Some simplifications were made in the model of the game, but these were mainly to remove complexities that would only serve to slow down the level generation process. A few features, such as lifts, were omitted because they could not be displayed graphically using the current layout used in the graphical user interface for the DGS.

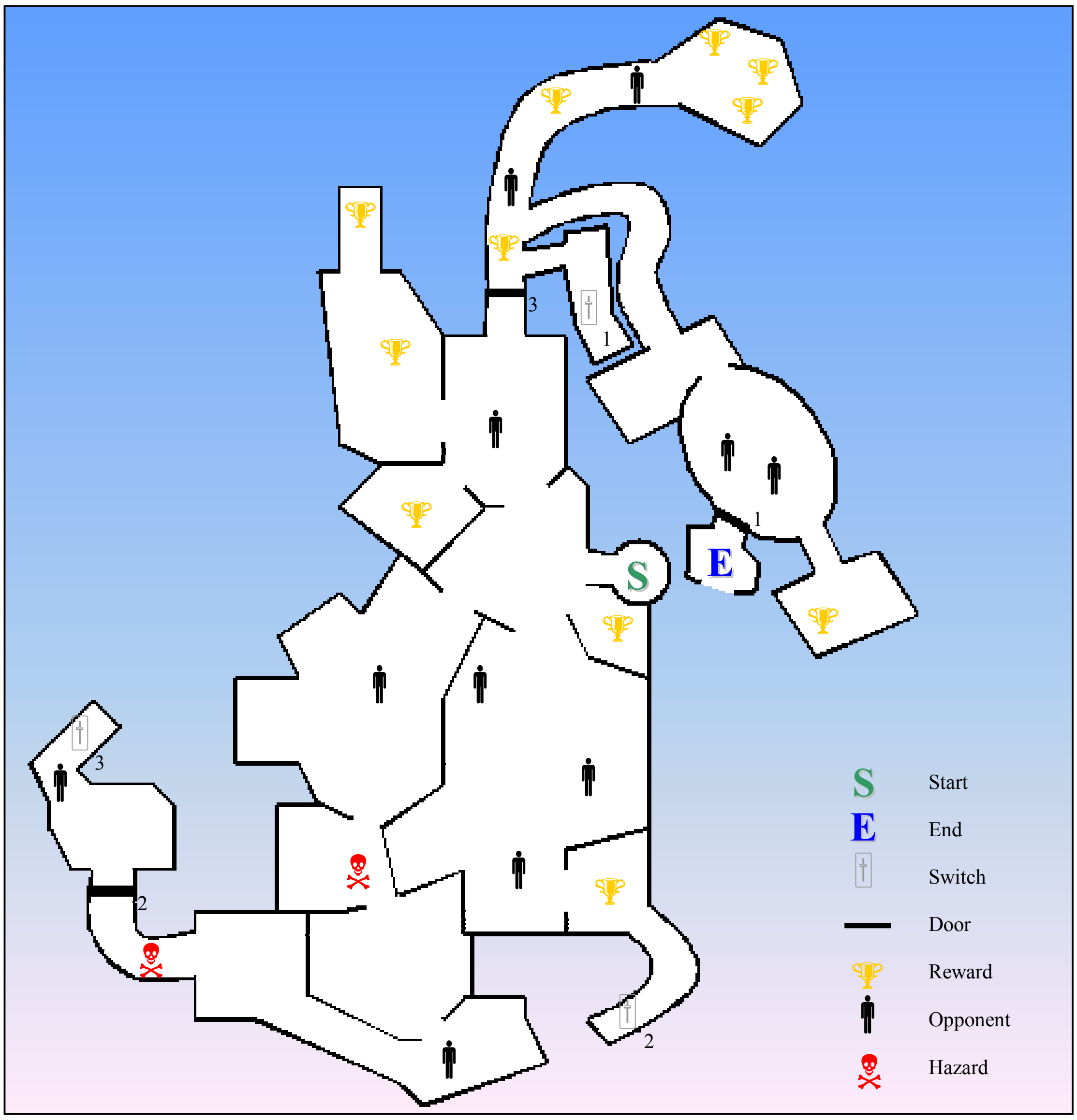


Figure 5.1.3b: A possible two-dimensional map for the level in Figure 5.1.3a

However, it was never intended for the simple layout of nodes that is currently available to be adequate to provide geometric descriptions of levels. It was merely created to help visualise the results attained during this project, as well as providing the basic infrastructure for a proper two-dimensional geometric layout of a level to be created in a future project.

This example has also shown that quality levels can be produced in which pointless areas have been dealt with. The ParameterEstimator objects created show how the extra information available about a specific game can be used to more accurately assess factors such as the difficulty and fun-value of a level. It is felt that the difficulty estimator produced is quite accurate, due to the approach taken of simulating a player playing the level. Any estimate of the fun-value of a level was always likely to be less accurate because the fun of a game is more subjective and depends on things not modelled by the system, such as the graphics used and the actual implementation of objects such as opponents.

Of course, it is not the Dungeon Generation System that is the source of this quality, rather the production rules and ParameterEstimator objects created for a specific game. However, the Dungeon Generation System provides a flexible and supportive framework that allows such production rules and ParameterEstimator objects to be created.

5.2 Efficiency

5.2.1 The Problem

During tests, it was found that one problem with the system was its inefficiency, in terms of the time it took to output a level. Due to the random nature of rule application, it is difficult to calculate the exact growth rate in time to output a level as the level size increases, but this is estimated to be exponential.

The time taken to output a level starts to become unacceptable with levels of fifty nodes or greater, although the exact time can vary widely based on the precise structure of the level. One source of inefficiency lies in the algorithm to deal with pointless areas. This needs to find all paths through the level, and this process can involve a lot of work, particularly when some activators and responders have already been placed (increasing the number of possible paths). The rule application algorithm in the Graph Grammar System is also inefficient, because it finds all possible matches for a rule and then selects one at random. If the graph is large and there are many nodes on the left-hand side of the production rule then the number of possible combinations to be examined can be thousands.

Changes were made to the Graph Grammar System and the Dungeon Generation System to deal with the problems listed and therefore improve the efficiency of the system. These are described in the following subsections.

5.2.2 Improving Rule Application

To improve the speed of graph application, a new method called findFastMatch() was added to the Graph class. This performs the same function as the original application method, findMatch(), but is more efficient.

It may be asked why the original rule application method worked out all combinations before selecting one randomly. Firstly, to create random graphs, the chosen match for a rule had to be random, so the first match that was found could not simply be returned. Secondly, the process of finding a match is split into sections, such as matching the nodes, then checking for edges and attributes. Therefore, more than one candidate match needed storing because some could be found to be invalid at a later stage, such as because of not having the required edges.

The findFastMatch() method works by returning the first match that it finds, but to ensure that the match is random, the list of nodes is randomly sorted before matching takes place. It uses a set of pointers to store which node combinations have been tried so far, so that if a match fails at a later stage, such as if the attributes of the matched nodes did not match with those in the production, then the algorithm would know what combination of nodes to try next.

5.2.3 Improving Dealing With Pointless Areas

The algorithm to deal with pointless areas in a level works well, but has exponential growth complexity. The problem is that to make sure that a responder blocks all possible paths through the level (i.e. the player cannot simply walk around it by taking a different route) so that the player must pick up the related activator, the algorithm finds all possible paths through the level.

For all nodes n on the primary path it tries to find another path that goes from n to the end node without travelling along the edge from n to the next node in the primary path. This involves a lot of searching. Also, if it finds other paths, the system also needs to check whether there are any alternative paths to these new paths.

To solve this problem, a new strategy to generate levels, called LevelStrategy3, has been created. This also uses the faster rule application algorithm discussed in Section 5.2.2. LevelStrategy3's algorithm to remove pointless areas works by temporarily altering the graph so there is only one path through it. Then, it does not need to search for other paths through the level because there will be none.

There are two sources of multiple paths through a level – loops and different ways of picking up multiple activators. It was realised that any paths that only differ in the order in which activators are picked up can be ignored because the same set of nodes will be visited on each of the paths. Therefore any responder that successfully blocks off one of the paths will also successfully block off the other paths as well.

This reduced the problem to how to temporarily alter the graph to remove loops from it. This was done by introducing the concept of a *super node*, as shown in Figure 5.2.3a. A super node is basically a node that contains a set of nodes that formed a loop. Before LevelStrategy3 tries to deal with pointless areas, it searches the graph for loops. If it finds one, it deletes all the nodes in the loop and replaces them with a super node that has the same edges as the nodes in the loop. This is repeated until the graph has no loops.

After pointless areas have been dealt with, the graph is returned to its previous form by expanding all super nodes back into the sets of looping nodes that they represent. As in LevelStrategy2, object productions are then applied to the graph, and the finished level is returned.

Of course, finding all loops in a level involves some search itself, although not nearly as much as finding all paths through a level. For some sets of production rules, such as the set in Example 4.1.3, loops cannot occur due to the nature of the rules. Therefore, LevelStrategy3 takes an additional parameter `loopsPossible`. This parameter is set to false by a user of the system if the production rules available cannot generate loops. This stops time being spent searching for loops if there will definitely be none to find.

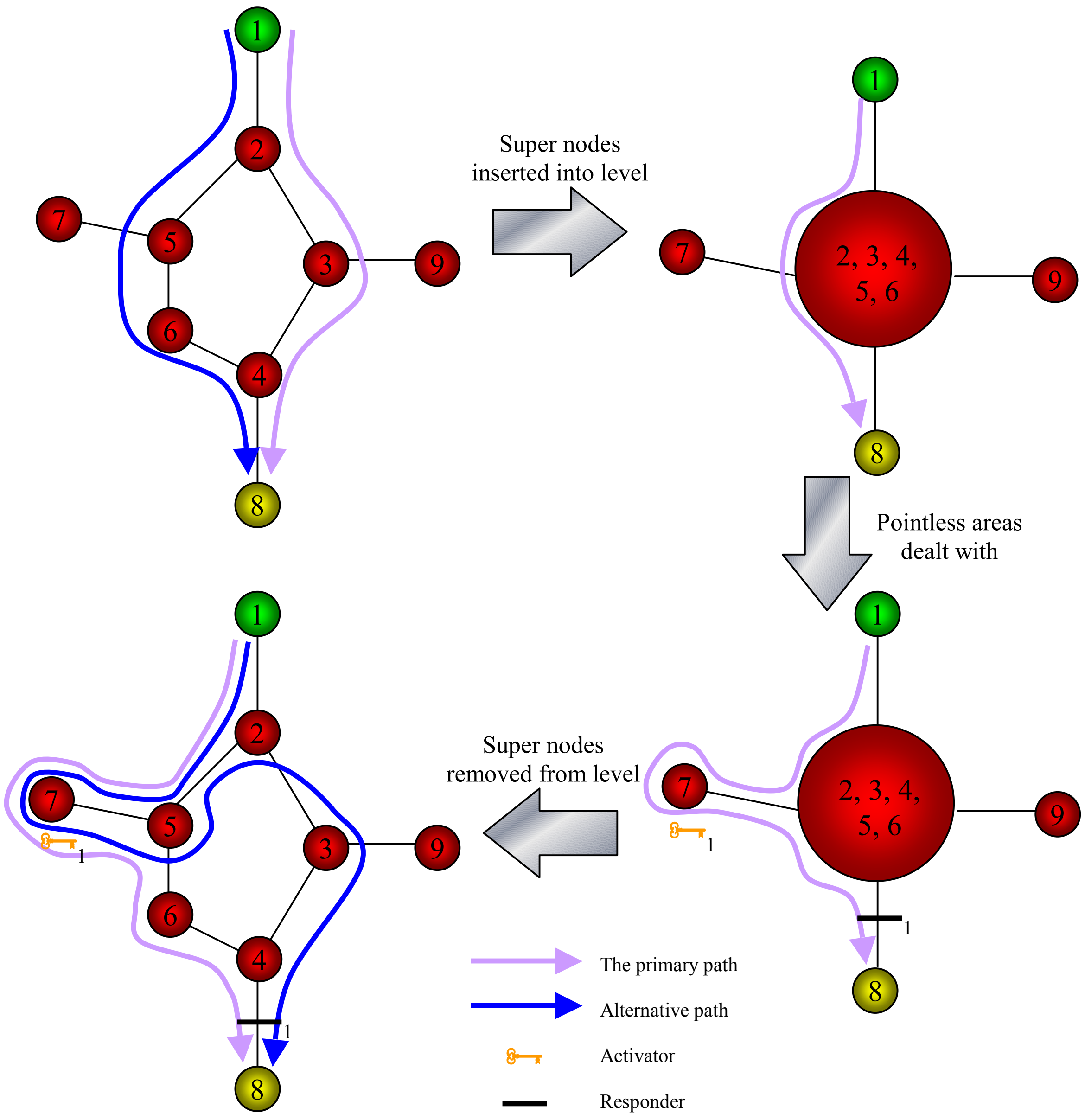
A less significant factor in the high complexity of the algorithm to deal with pointless areas is finding the primary path through the level. Calculating this once is not a problem, but the system recalculates it every time activators are added to a level, which can take some time when the level is very large. LevelStrategy3 takes another parameter, called `recalculatePath`, that when set to false means that the system does not recalculate the primary path after calculating it the first time. This leads to there being less variation in the placement of responders (because the primary path is not being recalculated to reveal other edges they could be placed on) but it speeds up the algorithm. It is up to the user to decide whether they go for larger variation or shorter time to create levels when setting this parameter.

5.2.4 Results

Tests were carried out to assess whether and by how much the measures listed in Sections 5.2.2 and 5.2.3 improved the efficiency of the system. This was done by assessing the average time taken by LevelStrategy2 (not using the efficiency measures) and LevelStrategy3 (using the efficiency measures) to create levels of varying sizes.

A program was written to conduct the experiment. In each case, the level that was requested to be generated had two parameters. The target value of the room size parameter (size of the level before dealing with pointless areas) was set to one of six values - 15, 20, 25, 30, 35 and 40. There was also a size parameter (the size of the level after objects have been put in it), and the target value for this was set to 250 in all cases.

The program requested ten levels from both LevelStrategy2 and LevelStrategy3 for each of the six values of the room size parameter used in the experiment. Thus, 120 levels were generated in total. The



1
2
3
4
5
6
7
8
9
Super nodes inserted into level
2, 3, 4, 5, 6
Pointless areas dealt with
Super nodes removed from level
The primary path
Alternative path
Activator
Responder

Figure 5.2.3a: An example showing how super nodes are used in level generation

system recorded the time it took to generate each level and displayed the results after all levels had been generated. These results are presented in Figure 5.2.4a.

The results of the experiment were much better than had been expected. The changes have seemingly reduced the growth rate of the level generation algorithm from exponential in LevelStrategy2 to linear in LevelStrategy3. The mean time taken to generate a level of target size 40 appears to be around twenty times less for LevelStrategy3 than with LevelStrategy2, dropping from around forty seconds to two seconds.

However, these results are slightly misleading. The median difference between the results with LevelStrategy2 and LevelStrategy3 is only about 5 to 10 percent, but the time taken using LevelStrategy2 is much more variable with levels occasionally taking as long as six minutes to produce when the target room size value was 40. This is in great contrast to the results for LevelStrategy3 where the difference between the minimum and maximum times for target room size 40 was only 691 milliseconds.

Therefore it can be said that LevelStrategy3 is much more consistent than LevelStrategy2 in that it always takes approximately the same time to generate a level of the same size. This is very useful for users of the system because they can estimate with some accuracy how long it will take to produce a level, instead of knowing that it could take anywhere in the range from two seconds to six minutes.

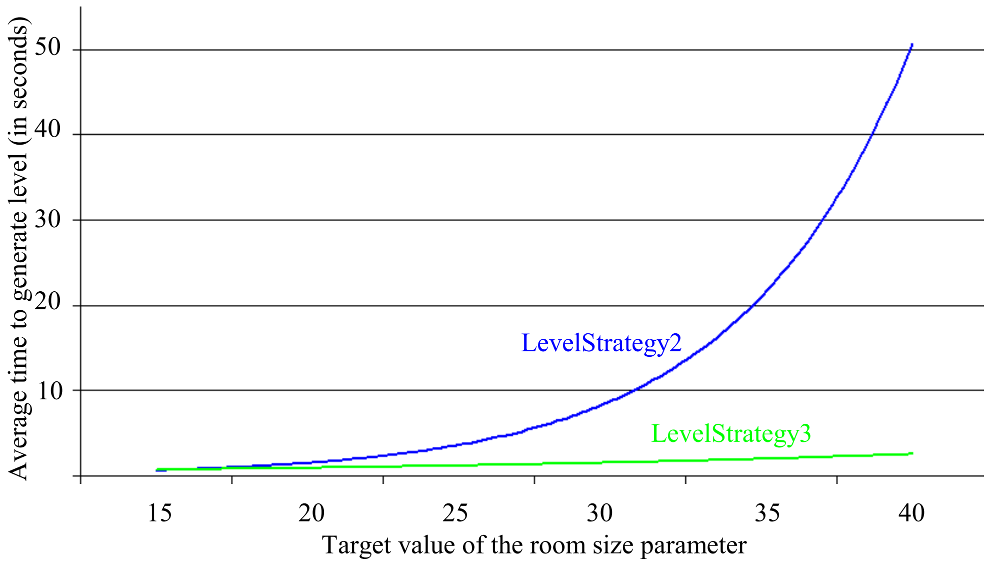


Figure 5.2.4a: Graph showing the relationship between target room size and average time to generate a level. Lines have been smoothed to highlight general trends.

Also, it was these rogue large times to generate a level that were the problem with LevelStrategy2, not the lower times of around two seconds, which are perfectly acceptable. Overall, it can be seen that these alterations to the program have significantly improved its efficiency, especially in reducing the worst-case time complexity, although there are other sources of slowness.

For example, the simulation of playing the game involved in the ParameterEstimator objects GameDifficultyEstimator and GameFunEstimator (see Section 5.1.2.4) slows down level generation significantly because a simulation needs to be done every time a rule is to be applied, and each simulation is computationally expensive to perform. This highlights the constant trade-off in level generation between quality (in this case, quality of the estimation of the difficulty and fun-value of a level) and speed.

5.3 Other Factors

There are a number of other factors on which the Dungeon Generation System can be evaluated. It is a highly flexible system for a number of reasons. Firstly, there is flexibility in matching conditions because matches are specified by Java methods in which anything can be written. Also, new ParameterEstimator objects, node types, level objects and strategies for generating levels can all be developed by the user and added to the system.

The graphical user interface helps to make the Dungeon Generation System easy to use. The system can also be said to have good scalability because additional production rules can be added without modifying any other files, and they are automatically available to be used in level generation. Finally, the system exhibits good modularity by the fact that the code to allow the manipulation of graphs (the Graph Grammar System) and the code related to generating dungeons (the Dungeon Generation System) have been placed in separate packages, leading to a separation of concerns. The Graph Grammar System says nothing about dungeons and therefore can be used in many other contexts.

There are, however, a few limitations with the Dungeon Generation System. Firstly, the search strategy employed could be said to lack focus and therefore wastes time trying options that could not possibly help it to produce a level that met the requirements. For example, under no circumstances will removing an opponent increase the difficulty of a level, yet the system still might apply such a production rule only to find that it of course does not produce a level that is closer to the requirements.

Secondly, the system does not have true data portability because the layouts of levels stored by the system can only be read by Java programs. However, the textual descriptions of the level are stored in a format that any program (or indeed human) could use.

Chapter 6 - Conclusion

In this conclusion, the main findings and achievements of the project will be summarised and it will be assessed whether the project was a success or not. The suitability of graphs for modelling levels will be discussed, and further areas of study will be suggested.

6.1 Project Summary

The project began with a thorough literature review. The genre of computer games known as dungeon games was described, as were previous attempts at generating content for computer games, such as fractal terrain generation. Graph grammars, the underlying mechanism used to generate levels, were also discussed, and it was found that no currently available graph grammar tool was available that was flexible and random enough to be used in the project. Lastly, game theory and psychology were examined to uncover principles of good level design.

Due to the fact that no suitable graph grammar tool was found, the first task in the project was to develop a system that allowed graphs to be manipulated, called the Graph Grammar System. It was implemented as a Java package and designed to be flexible so that any strategy can be created in Java and used to guide the process of rule application. The system also applies rules randomly, which helps to generate random levels. During the development of the Graph Grammar System, it was also observed that context-free graph grammars were very restrictive in terms of the languages of graphs they could generate, and that context-sensitive graph grammars were necessary to generate nontrivial graph languages.

To generate levels for dungeon games, another Java package called the Dungeon Generation System was created. This enabled users to input parameters representing features of the level they wanted output, such as its size, difficulty and fun-value. A number of general ParameterEstimator objects, used to estimate the value of a certain parameter in a level, were developed that could be used in any dungeon game.

Next, attention was turned to creating quality levels that did not have pointless areas, which are areas of the level that it would not be worth the player visiting in order to complete the level. This was achieved by finding pointless nodes in the level and inserting rewards and activators here to either force the player to visit these areas or make it beneficial for them to do so. The Dungeon Generation System was completed by developing a graphical user interface to make it easier for users to enter parameters for level generation and also to lay levels out graphically to help visualise the results.

Lastly, the system was evaluated on a number of factors. The quality of levels that could be produced by the system was assessed by producing a set of rules, node types, level objects and ParameterEstimator objects that modelled the behaviour of the real dungeon game Dark Forces [25]. It was found that varied, interesting levels could be created and that parameters such as difficulty could be measured more accurately using the game-specific ParameterEstimator objects.

The algorithms to apply production rules and deal with pointless areas were found to be inefficient. New methods were added to the system to provide more efficient implementations of these functions. Tests showed that these improvements reduced the average time to create a level, particularly for the worst-case scenario, and also greatly reduced the variation in times for levels of a certain size.

6.2 Success of the Project

It is felt that the project has been very successful. In the introduction to the project, a number of aims were set, and these have all been achieved. In many ways, these aims have been exceeded, because it was not originally planned at the outset of the project for a general system to allow graph transformation to need to be created; the plan was to use a tool that already existed. In addition, the graphical user interface for the Dungeon Generation System was not one of the initial aims and neither was the first step that was made in laying out levels graphically.

The two optional aims of the project, namely to develop a system to convert the topological level descriptions into geometric level descriptions and to create complete and playable levels for an actual dungeon game using levels output by the Dungeon Generation System, were not accomplished. It was realised during the course of the project that creating geometric descriptions of levels is a major task that would involve enough work to constitute a separate project. However, the layout of graphs that was created is a simple geometric description of a level, so the implementation of this can be seen as the first step towards creating more complex and useful geometric descriptions of levels, such as two-dimensional maps.

Also, creating even one playable level for a real dungeon game would require a large amount of work, and the benefit gained from such an exercise, namely that the estimates of size, difficulty and fun-value estimated by the DGS could be compared to those experienced by a player playing the game, was felt to be too small to justify such a large effort.

Furthermore, the project could also be assessed by examining how it compares with the work done by Simon Ince in the dissertation that this project was based on [21]. Ince used context-free grammars to generate levels, where terminal symbols represented objects in the level such as an opponent. It can be seen that in this project the following advances have been made: -

- Levels are always produced with the size and difficulty required. This was not always the case in Ince's system.
- Ways of estimating the value of parameters such as size and difficulty are much more sophisticated in the DGS, including path information and even simulations of playing the game.
- The grammar used in Ince's system had only two rules associated with the start symbol, both of which were highly detailed and specific. This led to levels being rather similar to each other. In this system, success has been found by starting from one simple node layout and using simple rules, such as a rule to just add a single node. A large number of applications of these simple rules have enabled greatly varying levels to be produced.
- The flexibility of rules in the system has meant that a smaller, more manageable set of rules is all that is needed to generate levels, compared with the larger set needed in Ince's system. This is because each rule can have multiple variations coded into it, such as a rule that can add a random number of opponents to a node. In Ince's system several rules would be needed for this, each corresponding to rewriting an OPPONENT_GROUP nonterminal symbol to a different number of opponent terminal symbols.
- Ince's system was not able to deal with multiple keys and doors because they were just represented by identical terminal symbols. In this system, activators and responders are Java objects and therefore each has a unique identity, which allows them to be distinguished from one another.
- The levels generated by the Dungeon Generation System are of better quality because pointless areas in them have been dealt with.
- In Ince's system, the only parameters that could be used to guide level generation were size and difficulty, and these could only be calculated in a fixed way. In the Dungeon Generation System, a parameter representing any feature of a level could be used, and the user can create their own ways of measuring these parameters.
- The string representations of levels produced by Ince's grammar only deal with objects, such as health packs and opponents. In the Dungeon Generation System, a graph represents both objects *and* physical areas such as rooms. This should help greatly when converting a graph into a geometric representation of the level it represents.
- Ince found that string grammars were not able to cope with complex corridor designs, since they are linear in nature. Graphs, on the other hand, are a natural way of representing such complex structures because a node in a graph can be connected to as many other nodes as is required. This allows more interesting levels to be produced.

From the list above it can be seen that major advances have been made over the work by Ince. The only advantage that Ince's system has is that it can produce levels faster. This is because rule application in a context-free string grammar is faster than in a context-sensitive graph grammar, because more work needs doing in such graph grammars to find a match. Also, extra features in the DGS to improve the quality of levels, such as dealing with pointless areas, take time and were not present in Ince's system.

6.3 Future Work

This project demonstrates that graphs and graph grammars provide a more natural way to model levels. Graph grammars combined with the flexibility of Java methods provide all of the features necessary to be able to represent even complex structures, such as rooms containing many different types of objects. They give a user of the system control over the derivation process that was lacking with simple string grammars.

Therefore, it is felt that it would be a waste of time researching alternative techniques to represent levels. It is also felt that using a different type of graph grammar system would also be unproductive. Attributes of rooms and corridors, such as the number of opponents in them, could also be represented using a hierarchical graph grammar [50] in which nodes representing rooms and corridors were themselves graphs, and these internal graphs contained nodes that specified the contents of that node. However, modifying the contents of a room node would then require the use of further graph transformations, which are slower than simply modifying normal attributes of a Java class. Therefore, it is believed that the type of graph grammar system used is the best for generating dungeons.

In Section 5.3, it was mentioned that the search strategy used in the Dungeon Generation System was unfocused. Perhaps a more goal-oriented strategy could be used instead. In this, the system would have a goal in mind, such as making the level more difficult. It would have some knowledge as to what goals each production rule could be used to achieve, and then would only try applying each of the rules that may help it achieve its current goal.

For example, it would be recorded that a rule that added an opponent to a level would help achieve the goal of increasing the difficulty of a level, but should not be used if the goal was to decrease the difficulty of a level. Such an approach would increase the efficiency of the system, because only the rules that were likely to produce a level nearer to that required would be tried.

A similar idea is to enable the system to plan several rule applications at once. Currently, the Dungeon Generation System has no plan; it merely reassesses the situation after each derivation. This may cause difficulties when target parameter values are difficult to achieve. For example, if the system must increase the difficulty of a level but keep the size constant, then it would be helpful if the system could formulate this in a plan, so it knew that even though adding an opponent (to increase level difficulty) may temporarily increase the size of the level, this was acceptable if it made sure that a later rule to be applied would reduce the size, such as by removing a health pack.

The above ideas could be tried, but they are rather minor points. The main area for future work lies in creating geometric descriptions of levels from the topological descriptions output by this system. The first stage of this could be to create simple two-dimensional maps where rooms were always square and the same size, and levels could not contain loops. Then the system could be improved so that it generated levels with rooms of varying sizes, for example, taking into account how many objects were in each room when deciding on its physical size. It is foreseen that creating a system to generate geometric descriptions of levels may prove a challenging task, but also a very rewarding one.

6.4 The Future of Dynamic Content Generation in Games

This conclusion will finish off by taking a look at the role that dynamically created levels may play in games in the years to come. As was seen in Section 2.2.2, in some game genres, levels are already randomly generated at runtime, such as in the strategy games *Sim City 3000* [31] and *Civilisation II* [36], and it is likely that many future games will also generate levels randomly.

However, random level generation for dungeon games is much harder, because a whole three-dimensional world needs creating, certain objects such as keys require careful placement, and more attention must be paid to managing the difficulty of the level. In addition, as these games develop, dynamically creating levels for them becomes a harder and harder task. Nowadays most dungeon games have storylines that may be disrupted by random levels, and as graphics improve and levels become more complicated, the level of sophistication at which any random level generator must operate, increases. What might be easier is for a random level generator to create levels at design time, which can

Figure 6.4a: Random levels increase the replayability of a game making it more popular with game players, but they might make users less likely to purchase add-on packs and sequels. Screenshot from *Civilisation II* [36]

then be improved by level designers before being put into the final game.

The main benefit for the user of randomly generated levels (at runtime, not design time) is that the *replayability* of the game is greatly improved. This means that randomly generated levels greatly increase the number of possible scenarios that the player can face.

However, increasing a game's replayability can also have possible disadvantages for developers, as described in [2], [3] and [11]. If a game is so replayable that people enjoy playing it no matter how many times they have completed it, then they might be less likely to purchase add-on packs and sequels, which provide game developers with a significant amount of their revenue.

To summarise, dynamically creating levels can bring benefits for both developers and game players, such as reducing development costs and increasing the replayability of a game. However, such random level generation is difficult to achieve for many genres of games, including dungeon games, and may in fact decrease sales of future releases. Therefore, it is felt that computer game levels will be created manually for some time to come.

References

[1] Adams, E., (1999). *A Letter from a Dungeon* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/designers_notebook/20000126.htm. [Accessed 01 August 2001].
[2] Adams, E., (2001). *Replayability Part 2: Game Mechanics* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/20010703/adams_01.htm. [Accessed 01 August 2001].
[3] Adams, E., (2001). *Replayability, Part One: Narrative* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/20010521/adams_01.htm. [Accessed 01 August 2001].
[4] Andries, M., et al, (1999). *Graph Transformation for Specification and Programming.* Science of Computer Programming, 34:1-54.
[5] Badcoe, I., (2001). Personal communication.
[6] Bleszinski, C., (2000). *The Art and Science of Level Design*. 2000 Game Developers Conference Proceedings.
[7] Blizzard, (1998). *Diablo*. Available from: http://www.blizzard.com/. [Accessed 09 December 2001].
[8] Blizzard, (2000). *Diablo II*. Available from: http://www.blizzard.com/diablo2/. [Accessed 09 December 2001].
[9] Chen, S., Brown, D., (2001). *The Architecture of Level Design* [Online]. Gamasutra. Available from: http://www.gamasutra.com/resource_guide/20010716/chen_01.htm. [Accessed 01 August 2001].
[10] Chown, T., (1997). *Diablo* [Online]. Games Domain. Available from: http://www.gamesdomain.com/gdreview/zones/reviews/pc/jan97/diablo.html. [Accessed 05 November 2001].
[11] Chown, T., (2000). *The Case for Random Maps* [Online]. Games Domain. Available from: http://www.gamesdomain.com/gdreview/depart/nov98/rmg.html. [Accessed 05 November 2001].
[12] Cormen, H., Leiserson, C., Rivest, R., (1994). *Introduction to Algorithms*. The MIT Press.
[13] Duvall, H., (2001). *It's All in Your Mind: Visual Psychology and Perception in Game Design* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/20010309/duvall_01.htm. [Accessed 01 August 2001].
[14] Ehrig, H., Engels, G., Kreowski, H-J., Rozenberg, G., (eds), (1999). *Handbook of Graph Grammars and Computing by Graph Transformations Vol 2: Applications, Languages and Tools*. World Scientific Publishing.
[15] Fournier, A., Fussell, D., Carpenter, L., (1982). *Computer Rendering of Stochastic Models*. Communications of the ACM, June 1982.
[16] Gallear, M., (1998). *Mark Gallear's Game Design Page* [Online]. Available from: http://www.geocities.com/TimesSquare/Arena/8461/index.html. [Accessed 05 November 2001].
[17] Hopson, J., (2001). *Behavioural Game Design* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/20010427/hopson_01.htm. [Accessed 05 November 2001].
[18] id Software, (1992). *Wolfenstein 3D*. Available from: http://www.idsoftware.com/killer/vintage.html - wolf3d. [Accessed 28 November 2001].
[19] id Software, (1993). *Doom*. Available from: http://www.idsoftware.com/killer/doommac.html. [Accessed 28 November 2001].
[20] id Software, (1996). *Quake*. Available from: http://www.idsoftware.com/quake/. [Accessed 29 November 2001].
[21] Ince, S., (1999). *Automatic Dynamic Content Generation for Computer Games*. University of Sheffield.
[22] Infogrames, (1999). *Unreal Tournament*. Available from: http://www.unrealtournament.com/. [Accessed 29 November 2001].
[23] Karanko, S., (1998). *Ideas and Algorithms for Realtime Fractal Terrain Generation* [Online]. Helsinki University of Technology. Available from: http://www.hut.fi/~samu/compgrap/birdland.htm. [Accessed 05 November 2001].
[24] Levine, D., (2001). *What is Game Theory?* [Online]. Department of Economics, UCLA. Available from: http://levine.sscnet.ucla.edu/general/whatis.htm. [Accessed 05 November 2001].
[25] LucasArts, (1994). *Dark Forces*. Available from: http://www.lucasarts.com/products/darkforces/splash.htm. [Accessed 28 November 2001].
[26] LucasArts, (1997). *Jedi Knight*. Available from: http://www.lucasarts.com/products/jediknight/splash.htm. [Accessed 29 November 2001].
[27] Marquis, J-P., (1997). "Category Theory". *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab.
[28] Martin, J., (1997). *Introduction to Languages and the Theory of Computation*. Second Edition. McGraw-Hill.
[29] Martin, K., (2000). *Using Bitmaps for Automatic Generation of Large-Scale Terrain Models* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/20000427/martin_pfv.htm. [Accessed 05 November 2001].
[30] Martz, P., (1997). *Generating Random Fractal Terrain* [Online]. Game Programmer. Available from: http://www.gameprogrammer.com/fractal.html. [Accessed 05 November 2001].
[31] Maxis, (1998). *Sim City 3000*. Available from: http://simcity.ea.com/us/guide/. [Accessed 28 November 2001].
[32] Maynard Smith, J., (1982). *Evolution and the Theory of Games*. Cambridge University Press.
[33] McCain, R., (1997). *Game Theory: An Introductory Sketch* [Online]. Available from: http://william-king.www.drexel.edu/top/eco/game/game.html. [Accessed 05 November 2001].
[34] Mehlhorn, K., (1984). *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag.
[35] Mendler, M., (2000). COM234 lecture notes. University of Sheffield.
[36] MicroProse, (1996). *Civilisation II*. Available from:

http://www.activision.com/games/civ2psx/. [Accessed 29 November 2001].
[37] Miller, G., (1956). *The Magical Number 7 Plus or Minus 2: Some Limits in Our Capacity For Processing Information*. Psychological Review 63: pp 81-97.
[38] Miller, G., (1986). *The Definition and Rendering of Terrain Maps*. SIGGRAPH 1986 Conference Proceedings.
[39] Minas, M., (1999). *Generating Semantic Representations of Diagrams*. Proceedings of the International Workshop on Applications of Graph Transformation with Industrial Relevance.
[40] Nash, J.F., (1950). *Equilibrium Points in N-Person Games*. Proceedings of the National Academy of Science of the United States of America 36, pp 48-49.
[41] Nash, J.F., (1951). *Non-Cooperative Games*. Annals of Mathematics 54, pp 286-295.
[42] Pfaltz, J., Rosenfeld, A., (1969). *Web Grammars*. International Joint Conference on Artificial Intelligence.
[43] Prestwich, K., (1999). *Game Theory* [Online]. Available from:
http://people.hws.edu/mitchell/hanley/games.pdf. [Accessed 05 November 2001].
[44] Rekers, J., Schürr, A., (1995). *A Parsing Algorithm for Context-Sensitive Graph Grammars*. Leiden University.
[45] Rekers, J., Schürr, A., (1996). *Defining and Parsing Visual Languages with Layered Graph Grammars*. Leiden University.
[46] Rich, E., Knight, K., (1991). *Artificial Intelligence*. Second Edition. McGraw-Hill.
[47] Ross, D., (1997). "Game Theory". *The Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab.
[48] Roth, A., (1991). *Game Theory as a Part of Empirical Economics*. Economic Journal, Volume 101, pp 107-114.
[49] Rouse III, R., (2001). *Game Design: Theory and Practice*. Wordware Publishing.
[50] Rozenberg, G. (ed.), (1997). *Handbook of Graph Grammars and Computing by Graph Transformations Vol 1: Foundations*. World Scientific Publishing.
[51] Rudolf, M., (1998). *Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching*. TU Berlin.
[52] Ryan, T., (1999). *Beginning Level Design, Part 1: Level Design Theory* [Online]. Gamasutra. Available from:
http://www.gamasutra.com/features/19990416/level_design_01.htm. [Accessed 01 August 2001].
[53] Ryan, T., (1999). *Beginning Level Design, Part 2: Rules to Design By and Parting Advice* [Online]. Gamasutra. Available from: http://www.gamasutra.com/features/19990423/level_design_01.htm. [Accessed 01 August 2001].
[54] Saltzman, M., (1999). *Game Design: Secrets of the Sages*. Macmillan.
[55] Schneider, H., (1970). *Chomsky-Systeme für partielle Ordnungen*. Technical Report 3,3, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen.
[56] Schürr, A., (1998). *PROGRES for Beginners*. The University of Aachen.
[57] Sierra, (1998). *Half Life*. Available from:
http://www.sierra.com/games/half-life/. [Accessed 29 November 2001].
[58] Smith, J., (2000). *Construction Complete? Computer Gaming's Battle to Take Over the World*. Future Publishing Ltd.
[59] The Columbia Encylopedia, (2001). *Theory of games*. Sixth Edition.
[60] TU Berlin, (1997). *AGG*. Available from:
http://tfs.cs.tu-berlin.de/agg/. [Accessed 29 November 2001].
[61] University of Bremen, (1998). *Treebag*. Available from:
http://www.informatik.uni-bremen.de/theorie/treebag/. [Accessed 25 March 2002].
[62] University of Aachen, (1998). *PROGRES*. Available from:
http://www-i3.informatik.rwth-aachen.de/research/projects/progres/. [Accessed 29 November 2001].
[63] University of Erlangen, (2001). *DiaGen*. Available from:
http://www2.informatik.uni-erlangen.de/IMMD-II/Research/Activities/DiaGen/. [Accessed 29 November 2001].
[64] Von Neumann, J., Morgenstern, O., (1953). *Theory of Games and Economic Behaviour*. Princeton University Press.
[65] Voss, R., (1987). *Fractals in Nature: Characterization, Measurement, and Simulation*. SIGGRAPH 1987.