Search-based Procedural Content Generation as a Tool for Level Design in Games

by

Jesper Lundgren

LIU-IDA/LITH-EX-A--13/063--SE

2013-11-28
Final Thesis

Search-based Procedural Content Generation as a Tool for Level Design in Games

by

Jesper Lundgren

LIU-IDA/LITH-EX-A--13/063--SE

2013-11-28

Supervisor: Erik Berglund (LiU)
Examiner: Anders Fröberg
Search-based Procedural Content Generation as a Tool for Level Design in Games

Jesper Lundgren

ABSTRACT
The aim of this thesis is to evaluate the use of Search-based Procedural Content generation (SBPCG) to help a designer create levels for different game styles. I show how SBPCG can be used for level generation in different game genres by surveying both paper and released commercial solutions. I then provide empirical data by using a Genetic Algorithm (GA) to evolve levels in two different game types, first one being a space puzzle game, and the second a platform game. Constraints from a level designer provide a base to create fitness functions for both games with success. Even though difficulties with level representation make it hard for a designer to work with this technique directly, the generated levels show that the technique has promising potential to aid level designers with their work.

Author Keywords
Games; content generation; genetic algorithms; evolutionary computation; level generation; design tool.

INTRODUCTION
Video game production is now a big entertainment market with titles like Grand Theft Auto 5 reaching $1 billion in sales in only three days [25]. New funding solutions as Kickstarter has also helped founding many new games created by smaller teams of developers [28]. This shows that the gaming industry is very attractive and it seems to be growing still. The game creators want to provide a lot of content to make gamers interested in their game while also having additional content for sale as an added revenue stream. However, with more content there has also been rising costs associated with artists and developers to produce more content for games [24]. In order to reduce cost and bring more content into games, various tools have been developed to ease the difficulty with content creation. For example, the Steam digital distribution platform for games provides tools that allow users to create and publish content for some of their favorite games [35]. Even back in 1999, the game Age of Empires 2 provided a level editor for gamers so that they could create their own campaign scenarios and share them with other gamers [31]. Having users create content for games could have some disadvantages like making quality control difficult. This could mean there will be a lot of bad content in circulation making good content both difficult and bothersome to find for gamers [24]. Another way to get more content is to have the computer create content by following a set of rules, a technique called Procedural Generation (PG). This thesis will explore a subset of the PG technique called Search Based Procedural Content Generation (SBPCG) and how it can be used as a tool to create levels for games. The aim is to; provide a summary of use cases in different games and then provide an empirical evaluation by creating levels in two games using a Genetic Algorithm (GA) together with constraints.

PROCEDURAL GENERATION
PG has been put into practice with success to solve the problem of generating a lot of content in games as early as 1980 when the game rouge was released [33]. Rouge is a dungeon crawling game that uses graphical representation for dungeons. Although praised for its graphics one of the developers behind rouge said he thought the game’s biggest contributions was that it could generate new adventures [26]. A bit later in 1984 the space trading game Elite was released [29] where PG was used to generate all the planets in the galaxy [22]. PG can also be used as a tool to help users create higher quality content. In the game Spore, players could create their own animals by using an easy interface, while textures and animations were created with PG instead of having players do it manually [27]. When it comes to graphical content, many commercial tools that exist today implement the PG techniques as well, such as SpeedTree, which allows designers to create trees according to their own specification [34].

Procedural Content Generation
There should be a short mention about Procedural Content Generation (PCG) which is a definition used in some writings instead of PG. PCG refers PG for content that specifically affects game play, such as tools or similar needed to complete a mission. This is in contrast to generate textures, which affects the graphics but not necessarily the game play. PCG also aims to put a larger focus on randomness meaning that just small parameter changes should be able to produce a large variety of content [32].
**Offline/Online**
PG can be run in two different situations, “online” or “offline”. Online means that the PG will be used to create content in real time or near real time during gameplay. This could be either to add content continuously to a game or to provide new content during short load sequences in a game.

PG can also be used during the game development process, helping designers by generating content to their specification, which is called offline generation. The newly generated content could then be used as static content when the game is released. There are some requirements to consider for the PG algorithm depending on if it is to be used in online or offline settings. Online PG usually has to be much faster since it can affect game playing experience if it causes slowness to the game. The online setting also makes it difficult to verify the quality of the generated content, which means that the PG algorithm has to be more reliable. This becomes extra important when PG creates content that could be critical for a player’s ability to play the game. Offline PG on the other hand can take longer time, generating content for days or even months at a time. A human designer will be able to verify the content produced in offline setting before it reaches the end consumer, which means that the algorithm does not have to be as reliable [21].

**Search-Based PCG**
PCG can be done in two main ways, constructive or generate-and-test. Constructive methods such as fractals, Mandelbrot or similar are often adopted when speed and reliability of content generation is important [21]. An example of this is fast generation of trees and plants with fractal functions where a designer can edit some parameters to get different looking material [9].

A generate-and-test method in contrast to constructive methods verifies the validity of the generated content and will try to construct again if it does not meet some requirement [21]. SBPCG is a special case of the generate-and-test method. It utilizes a fitness function (evaluation function in some literature) to guide a search algorithm through a search space of content. Searching is common in many AI techniques, where exploring a decision space is a regular problem, similar to the problem of exploring a content space. In fact, most game developers are probably not a stranger to these search techniques; AI researchers have often made use of games as a research platform. They have for example been creating logical agents for the Wumpus World game or game playing AI using the minimax algorithm [12].

The generate-and-test nature usually makes SBPCG slower than constructive methods, and a larger space of content can affect the reliability of content quality. However, not all problems are easy to analyze a rule structure for (to create a constructive method from) and in these cases, the use of search techniques has been successful in other fields such as mechanical engineering, civil engineering and software engineering [5]. The use of a fitness function also provides other interesting additions like the ability to have a human guide the search in an interactive way to create user preferred content [14].

**LEVEL GENERATION OVERVIEW**
This section aims to provide an overview of how SBPCG has been used to partly or completely create levels in different game genres. This is done by mostly summarizing different research papers in the field of SBPCG where it has been applied to level generation. I have shortlisted some examples from five main game genres, Strategy, Adventure/Puzzle, Platform, First Person Shooter and Racing. There are of course more genres but I believe that these are some of the major ones and the techniques presented for these genres could be an inspiration for others. For a more thorough review of the SBPCG field covering more than just levels, I refer the reader to “Search-based procedural content generation: A taxonomy and survey” [21].

**What is a level?**
It is not always so easy to define what a “level” in a game is. A level is usually some way to pack game play in a small package that works within the rule set of the complete game. They are often some kind of space where there is some objective to solve. In a first person shooter, game levels usually consist of a new environment (“map”) and depending on the game there might also be a new mission or similar. For platform games, the environment and enemies usually change between levels but in tower defense games, the environment usually stays the same while new (usually more difficult) enemies arrive. Some games such as Angry Birds have many levels while others as DOTA arguably only has a single level. For games that use many levels it is also common that they differ in difficulty, making levels a way to provide difficulty progression in games.

**Strategy games**
Liapis,Yannakakis and Togelius [6] developed a level design tool that works by allowing a designer create low-resolution sketch of a strategy game map, with a similar look to the maps in StarCraft. As a designer is working on a map, the software is first evaluating the “playability” of the current sketch by checking that all resources and bases are reachable from all other resources and bases.

Once the playable criteria is met, the software evaluates the map according to six fitness dimensions such as “resource safety”, “exploration”, “safe area balance” and others. The dimensions are shown as a value or as an overlay to help the designer evaluate the current design.

In addition to allowing the designer to be able to manually edit the map, the tool also provides suggestions. These suggestions are created by using genetic search algorithms with the current map layout as the start of the search. There
is one search for each fitness dimension; each search provides a suggested design that improves the score for that dimension. A novelty search is used to give six alternative map layouts that are far apart from each other based on a model of what makes map layouts different. The goal for these suggestions is described as “challenge the user’s current design focus and provide unforeseen alternatives to achieving the designer’s goals” [6].

In a small user study conducted with five expert users, it is found that the designers often had specific ideas in mind when designing a new map, which meant the suggestions were not so useful. Another issue was that users often wanted to create a symmetrical layout so the generated suggestions were not interesting because they were more organic than symmetrical. As a way to improve some of the issues, the ability to “lock” certain features such as number of bases is discussed [6].

Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelback, and Georgios N. Yannakakis used a multiobjective evolutionary algorithm in order to generate interesting maps for the popular real-time strategy game StarCraft [20]. Maps are encoded using the main elements of interest such as base, resource and impassable area (mountains and rivers) positions, which are then added to a clean map with only passable area. Using the shortest distance between locations (calculated using A* search on the map), different fitness functions are suggested such as “base distance”, “resource ownership” and others. These functions are then matched in pairs to work with a multiobjective evolutionary algorithm. The algorithm will try to find solutions that optimize both fitness functions in order to generate a good StarCraft map. A discussion about conflicting functions, difficulty to optimize functions ends the paper together with arguments on which function pair produced the maps most similar to typical StarCraft maps.

**Adventure/Puzzle games**

David Pizzi, Jean-Luc Lugrin, Alex Whittaker, and Marc Cavazza developed a tool [10] to help level designers in multiple solution games like Grand Theft Auto, Metal Gear Solid or Hitman. These games aim to provide a player the choice of their own style and plan in order to solve different missions. The tool can find a sequence of actions that the player could take to solve a mission. This is done by creating a simplified description of states and actions within a level, and then use heuristic search planning to find solutions. The simplified description follows planning language structure, with states such as “location(X, room)”. With Hitman as example, the search could be modified to account for different play styles such as being more stealthy or going for more kills.

In order to help a designer visualize a solution, the tool also generates images into a storyboard, showing the solution sequence of actions. A designer could then modify the environment or the action sequence to see how it would change the plan or possibly make it unsolvable. By seeing

the complexity of a plan and the number of sub plans, a designer can evaluate the need to balance the difficulty.

The tool could sometimes produce “fragile” plans, which refer to the plans that are sensitive to timings or the use of resources in the game. The resulting situation could be that the plan fails because of changes in enemy movements or that a player uses up all ammunition in a weapon. Ways to mitigate these issues are discussed as future work in their paper [10].

Dart, Isaac, and Mark J. Nelson describe a way to increase replayability in adventure games by procedurally generating sub-puzzles in the game instead of generating a new story line [3]. Interactive items in a game can be chained together by how they can be used (action), what effects they have and what they can cause. For example, on a paper item, a player could take an action called burn. This action would cause a fire, which would have the effect of smoke. Such a chain creates a directed graph called Smart Terrain Causality Chain (STCC). Starting from an objective in a puzzle (like activate a sprinkler), a STCC can be generated and be solved like a planning problem using the properties of items in the game. By letting players try their experimental game called “Space Dust”, which uses this technique, Dart, Isaac, and Mark J. Nelson found that most players found multiple solution puzzles easier to solve but could be confusing. The length of the puzzles also had an effect on the difficulty.

**Platform games**

Nick Nygren, Jörg Denzinger, Ben Stephenson and John Aycock presents a system for generating levels in platform games that follows user preferences in their paper “User-preference-based automated level generation for platform games” [8]. For their system, they describe two phases, one for “user preference elicitation” and the other for “level generation”. In the first phase, they let a player answer multiple-choice questions in combination with measuring performance in short pre-defined levels, which creates a user preference model.

Then in the “level generation” phase, the information gathered from the first phase is used as design criteria for a three-step generation process. The first step in this process is to generate a directed graph to control the overall structure and large-scale complexity of the level such as how many times a player have to make a choice between different paths. This is done by using a feasible-infeasible two population genetic algorithm with constraints and fitness function that follow the user preference model.

The second step is to use the most suitable graph to create cells that represent the size and placement of smaller areas within the platform level.

The last step is to fill each cell with content such as enemies, coins or other components. Search is used to look through possible content compositions and stops when some playability constraints are being met and the content
suitable according to the user preference model. The system is tested with four users that have different preferences, showing how the system performs in generating levels with styles.

Gillian Smith, Jim Whitehead and Michael Mateas introduced a level design tool for platform games called Tanagra [15]. This tool can help a designer create a level by creating platforms that follow constraints created by the designer. A designer could later edit any platform and Tanagra will modify surrounding platforms, making sure the level is playable. Levels in Tanagra are created according to a “rhythm” on how often a player has to take a new action such as jumping, running or killing an enemy. The level generation in the Tanagra uses A Behavior Language (ABL) planning language to choose platform style based on a designer’s choice and then a constraint solver to decide the platform placement in the level.

In the platform games category, there are many level generating prototypes using different techniques. One reason for this is a competition where researchers compete in creating the best level-generating algorithm for a Mario style platform game [13].

There are also some commercial implementations like Cloud Berry Kingdom. One of the Cloud Berry Kingdom developers, Jordan Fisher, described in an article on Gamasutra [23] how he designed the system used for producing platform levels with different difficulty in that game.

**First person shooter games**

Luigi Cardamone, Georgios N. Yannakakis, Julian Togelius and Pier Luca Lanzi describe a way to generate interesting maps for a first person shooter game with evolutionary algorithms [1]. Using death match mode in the open source game Cube 2 as test platform, they develop a fitness function for a level based on a theory that longer engagement is more fun. The fitness function is also extended with a measure on how much free space there is on the map to make sure it is enough space to add weapons and spawning-points. They test four different ways of describing (encoding) the levels for the evolutionary algorithm. From a more direct encoding when it is closer to the level specification in the game, to a more indirect encoding when it only evolves parameters to a function that in turn generates the map. During each evolutionary step, the maps are tested in the game’s simulation mode with four bot players playing a death match. The engagement time from the match is then used in the fitness function. The use of simulation mode and bots is a faster alternative to having humans play each time. Some objections against this technique are raised, like that this measurement depends on the quality of the bots [1].

**Racing games**

Julian Togelius, Renzo De Nardi and Simon M. Lucas presented a way of using a player model to generate personalized tracks for a racing game [18]. By measuring players’ driving habits such as speed and positioning on test tracks, an AI can be created to represent the player’s driving style. Different approaches for creating such AI are tested and discussed.

A model of fun for racing tracks that uses a players driving speed together with how much a their progression varies on a track after five trials as key indicators for fun are proposed. The player representing AI drives on a proposed track while so the key indicators can be measured. These measurements are then used in a fitness function for an evolutionary algorithm in order to evolve tracks suited to a player.

Tracks are encoded as a sequence of b-spline curves in order to give smooth corners and an easy way to loop the track. Different ways of initializing the tracks for the evolutionary algorithm are tested in order to see the different curvature styles tracks could get when encoded as b-splines. This is also compared to the results they got from an earlier paper when they experimented with similar techniques but used a different way to represent tracks [19]. They suggest using the same techniques on a more advanced racing game together as future work while studying more on what players think is fun in racing games [18].

**GENETIC ALGORITHM**

A Genetic Algorithm (GA) is a search algorithm that uses techniques inspired by evolutionary biology such as sexual reproduction. These kinds of algorithms have had a widespread impact on optimization problems such as circuit layout or job-shop scheduling [12]. With GA, a possible problem solution is represented as an array that is similar to a chromosome found in natural evolution (for GAs solutions are most commonly represented as a single chromosome but in nature it could be many combined). The proposed problem solution might not be valid at first, but using evolutionary steps a GA could find a new solution that is valid.

**Evolutionary steps**

A GA will first initialize a group of possible solutions (most commonly referred to as a population within GA literature). Every individual in this population is then given a fitness score from a fitness function. The fitness score is used to select pairs of individuals for mating. The mating of two individuals will result in an offspring whose chromosome is a combination of the parents’ chromosomes. The offspring also has a chance of mutation, which means that a small piece of the chromosome might randomly change. An offspring is a new individual, and all new individuals become a new population that will have to follow the same
procedure. The three steps just described for creating new individuals are called selection, crossover and mutation. They are the genetic operators (or search operators) that help the population move in some direction of the search space with help from the fitness function.

**Representations**
There are many different kinds of representation that could be used for a chromosome. For example, it could be represented as a binary string, real valued vector or an integer vector. With these representations it is easy to apply the standard search operators, however not all problems can easily be encoded into those representations. Representation difference also occurs in nature were our genetic material does not directly map to our physical appearance. To be able to move from genetic material to physical appearance, there is a genotype-phenotype mapping, where phenotype represents our appearance. The search operators are working on the genotype representation while the fitness function evaluates the phenotype [11]. The distinction between genotype and phenotype within the subject of levels for games can be seen as the instructions for creating a level (genotype) and the actual level (phenotype) [21].

**Direct vs. Indirect**
The mapping between the genotype and the phenotype can be more or less indirect, and depending on the representation and mapping being used to solve a problem, the search space will change. With a direct mapping, there is no need to do any conversion from the genotype to phenotype, while with a very indirect mapping the genotype might only represent a seed to be used in an algorithm that can produce solutions.

With indirect mapping, the genotype representation is typically shorter, which means that the search space for the GA to explore is smaller. A smaller search space is faster to search but it is possible that not all solutions can be reached since there might not be a mapping for every solution.

A direct representation on the other hand typically has a larger search space since in order to represent some complex solutions directly in a chromosome, the representation vector usually has to be large. With direct representation it might even be needed to modify the search operators for offspring’s to be able to gain important properties from their parents [11,12].

The issues of mapping and representation together with the stochastic nature of the GAs (stochastic due to randomization during selection, crossover and mutation) make it difficult to do a detailed prediction of the performance and efficiency of the algorithm. GAs have been used with success in different areas mostly surrounding optimization. There is an active research community with IEEE Transactions on Evolutionary Computation journal in fourth place on impact factor ratings in the category of computer science/artificial intelligence among similar journals analyzed by Institute for Scientific Information (Thomson ISI). It is also believed to be more than 10,000 papers published in the evolutionary computation subject [4].

**MY WORK**
I have used a GA with a constraint based fitness function to create levels for a designer to work with. The intention is to find a way to reduce tedious and time-consuming tasks for a designer and to enhance the variety of the levels. I have tested this technique on two games of different game styles while I act as the level designer. This means that the constraints follow the properties of a level I wanted.

The constraints in the fitness function could be seen like rules a designer decides for a level. For example in a platform game a designer could decide how large/small a platform can be and how two or more platforms should relate to each other.

“Deep space”
The first game is called Deep Space and it is a newly developed game by me as part of this thesis. Deep Space is a space puzzle game where a player is supposed to move from start to goal without being noticed by enemy ships patrolling the space. How far an enemy can see is visualized as a circle around their ship, which is supposed to help a player to know where to doge. Each enemy has patrolling points that it follows repeatedly. The idea is that since each enemy follows a strict patrolling pattern (by moving to each point in order like check points), a player can recognize the predictable movement. The predictable movement will help a player find a path to the goal without being noticed. The task of spotting a path is intended to be the puzzle of the game. Deep Space is similar in style to “The worlds hardest game” [30]. Figure 1 displays a level in Deep Space. The green triangle without a circle is the space ship controlled by a player and the white square is the goal that a player should reach without being detected.

![Figure 1. A level in Deep Space](image)
In Deep Space, a level is stored as a text file with information on start and goal position, number of enemies and their respective patrolling points together with coordinates for a static object in order to confine a player to a smaller space.

“Friday”
The second game is a platform game called Friday, that is similar to Mario but with some different mechanics. I developed Friday together with Johan Eriksson in the course “Design and Programming of Computer Games” at LIU. In this game, a player should get from start to goal by jumping and running between platforms while watching out for obstacles that could hurt the playable character. These obstacles could be a ray of light that burns a player when in contact, spikes that kill a player on touch or falling boxes that could kill or just block the way. The main common theme for all levels is the use of platforms. Figure 2 illustrates one of the levels we created for Friday. The white box is the goal a player should reach by jumping over the dangerous spikes. A player can try to get the purple box before going to the goal if they want to make the level more difficult.

![Figure 2. One of the levels in Friday.](image)

There is no level editor for this game and no specific file to store levels in, so all levels are entered in the source code. Because of this, manually creating levels for Friday could take a lot of time and testing, especially if a lot of platforms or obstacles are needed.

Tools
Since the plan is to generate more of a template than a complete level, there is no need for the generation to be a part of the game code. My intention is to use the same tool for both games so I decided to look for some general tools to use for development. The first approach was to use octave and octave-ga package (GA framework) because it is an easy development environment for algorithms and has many geometric functions available that would be useful. However, when I had some trouble getting octave-ga converging for an early test fitness function I decided to look for alternative tools. Python together with PyEvolve (GA framework) became my final choice.

Python is a good environment for algorithm development allowing rapid development and testing together with being easy debug. PyEvolve, though being an inactive project has good documentation and is designed in a way that makes it easy to understand and extend if needed. More importantly, the early test fitness functions I had do converge with PyEvolve, which makes it a much better choice.

The switch to Python had some downsides as I lost the built in geometric functions and plotting tools. To replace this functionality I used Shaply for geometric functions and NumPy together with Matplotlib as my plotting tools.

Deep Space level generation
For Deep Space, I divided the level generation into two steps. The first step is to generate the points that form the patrolling pattern for an individual enemy, while the second step adds many enemies together making sure their pattern overlap and protect the goal. The two-step process allows a designer to intervene and do manual edits if it is desired. It also helps making the fitness function and representation simpler.

The patrolling points for an individual enemy is encoded as a matrix with x and y values as columns and one row for each patrolling point (x,y coordinate). In PyEvolve this is done as a part of the genome setup process which looks like this.

```python
import G2DList
from G2DList import G2DListMutatorIntegerRange, Initializators

# Genome setup
num_points = 15  # Number of patrolling points
rangemin = -300  # Range minimum
rangemax = 300   # Range maximum
bestrawscore = 0.00  # Best raw score
roundDecimal = 2   # Round decimal

genome = G2DList.G2DList(num_points, 2)
genome.setParams(rangemin=rangemin, rangemax=rangemax, bestrawscore=bestrawscore, roundDecimal=roundDecimal)
genome.mutator.set(Mutators.G2DListMutatorIntegerRange)
genome.initializator.set(Initializators.G2DListInitializatorInteger)
```

G2DList.G2DList is the genome representation being used, with the parameters deciding row and column size. The rangemin/rangemax parameters will limit the x/y values generated by the genome initialization and mutator.

One of the main parts of using a genetic search function is to create a fitness function for the problem to be solved. My first approach was to let the genetic algorithm maximize a fitness function that gave a score for each constraint being met. This means that in order to know when all constraints are being met one has to know the total score possible, which is not always so simple if the fitness function has many constraints and is complex.

Transforming the problem to a minimization problem simplifies this. Giving a score (penalty) every time when a constraint is broken means that a score of zero is a suitable level. This technique is similar to static penalty described Coello Coello, Carlos Artemio [2]. The penalty given for each broken constraint can be a measure on how badly the constraint was broken (that is, the distance to the constraint boarder).
First step
For the enemy patrolling points, I designed the patterns to be like a chain. To do this, I decided on three main constraints.

1. minimum distance between any point
2. maximum distance between any point
3. min/max distance between neighbor point in terms of their patrolling order.

For the first two constraints, I used this code.

```python
for p1_coord in chromosome:
    min_distance_count = 0
    max_distance_count = 0
    p1 = Point(p1_coord)
    linelist.append((p1_coord[0], p1_coord[1]))
for p2_coord in chromosome:
    p2 = Point(p2_coord)
    distance = p1.distance(p2)
    if distance < min_distance:
        min_distance_count += 1
    elif distance > max_distance:
        max_distance_count += 1

    score += max_distance_count
    score += min_distance_count
    score += 1 # self distance removed
```

Here I loop through all points and count for each one how many points either too close or too far away. The two counters are then added to the total score for the chromosome to be evaluated. Adding the counter instead of a fixed penalty works as a measure on how far away it is from meeting the constraint. I plot the best pattern currently found every time the algorithm has evolved the population a number of generations. This plot can help debug a fitness function and give a better understanding on how the search is progressing. Figure 3 shows how this plot looks like.

![Figure 3. Plot from ongoing pattern generation for 15 points.](image)

Second step
For the second step of the level generation, the genome is encoded by storing values representing x-displacement, y-displacement and rotation for each patrolling pattern. The fitness function used for this part applies two main constraints.

1. Percentage of overlap between patterns
2. Distance to the goal for each pattern

The assumption is that more overlap between patterns and closer to the goal create a more difficult level.

```python
for x in xrange(size):
    translated =
    translate(linestrings[x], chromosome[x][0], chromosome[x][1])
    modified.append(rotate(translated, chromosome[x][2]))
for line1 in modified:
    intersect_count = 0
    if goal.distance(line1) > goal_distance_limit:
        score += (goal.distance(line1) - goal_distance_limit)
    for line2 in modified:
        if line1.intersects(line2) and line1 != line2:
            intersect_count += 1
            if intersect_count < (size*min_limit):
                score += (size*min_limit) - intersect_count
            elif intersect_count > (size*max_limit):
                score += intersect_count - (size*max_limit)
```

Here the original patterns are first modified according to the values in the chromosome (can be seen as a genotype to phenotype translation) to then be evaluated on both distance to goal and how many times it crosses other patterns. The added score for each constraint uses the difference from the constraint value to give a measure of distance to the constraint to better guide the GA. Again, I use plots to help evaluate the progress and result of the algorithm, an example of this can be seen in Figure 4.

![Figure 4. Plot from the second part of deep space level generation. Seven enemies with the pattern length three.](image)
As a final touch to the level, a convex hull for all patrolling points and the player start position are used to create the static body surrounding the level.

Friday
For the platform-game Friday, I used the same techniques as with Deep space, but in this case limited it to only generate levels with platforms and let a designer add other obstacles.

In Friday, a platform is described with three values. The three values are, x,y values for the starting position plus a value for the width of a platform stretching out in positive x-axis. This same description is also used for the genetic encoding by having each row be a platform in the game, creating an easy and direct mapping to the game. An alternative way to encode the platforms would be to let each platform except the first one be described as a displacement from the platform in order, as is done by Nathan Sorenson and Philippe Pasquier in their paper "The evolution of fun: Automatic level design through challenge modeling." [16]. However, I think that this could limit some options in placements since the platforms become chained to each other in the representation.

Fitness function
I used three constraints to create the fitness function.

1. Minimum distance between platform
2. Number of platforms reachable from the ground (starting platforms)
3. All platforms should be reachable

For the first constraint, the following code is used.

```python
def level_path(start,platforms):
  traversed = []
  traversed.append(start)
  for platform in platforms:
    ... and platform[1] - start[1] < 6:
        traversed.extend(level_path(platform,platforms))
  return traversed
```

The second constraint in a similar way but is much simpler since it only needs to check the distance from the ground floor.

```python
if platform1[1] < max_jump_height:
  ground_platform_count +=1
  traversed.extend(level_path(platform1,platforms))
```

The count is then added to the total score.

The last constraint is to make sure that all platforms are reachable. The first platforms a player can reach are the ones close enough to the ground floor. I call these platforms "ground platforms". From each ground platform, a Depth-first search (DFS) finds all other platforms reachable by a player. The search uses a simplified model of the player physics to decide whether a platform is reachable, Nathan Sorenson and Philippe Pasquier also used this technique.

```python
def level_path(start,platforms):
  traversed = []
  traversed.append(start)
  for platform in platforms:
    if platform1_line.distance(platform2_line) < 3:
      distance_count += 1
  if distance_count > 1:
    score += distance_count
```

This code iterates over all platforms and counts for each one how many surrounding platforms that are too close. In this code the variable platforms is a list created from the chromosome. Every item in the list is an array describing positioning and width of a platform. This information can create a Shapely LineString object representing a platform, which can be used calculate the distance to other platforms. The total count for each platform added to the score in the same way as it was done for Deep Space.
Depth First Search

The DFS function I implemented works by storing all reachable platforms in the list variable `traversed`. The function begins by appending a known reachable platform called `start` to `traversed`. After this, it iteratively goes through all platforms comparing them to `start` in order to find a platform that is above and reachable.

I have divided the reachability check into two cases. The first case is when a platform is on either side of `start`. In this case, the simplified physics model is used to verify whether a player can reach the new platform from `start`.

The second case is when a platform is above without being beside `start`. In this case, a new platform is reachable from `start` if it is close enough while not completely covering `start`. In other words, there should be some space for a player to jump up without colliding with the new platform.

As soon as the function finds a new platform that is reachable it recursively continues the search from there by making it the new `start` platform. Since the search only considers platforms that are above the `start`, the platforms can be seen as a tree structure. If the search would also look for reachable platforms below `start`, it would become a graph structure. The graph structure could have loops causing this implementation to be stuck searching forever.

All reachable platforms will be stored in `traversed` in the end. This fact can then be used to create a penalty for every platform that is unreachable (not available in `traversed`). With this implementation, platforms that were reachable multiple times are stored as duplicates in `traversed`, making it possible to add a constraint that ensures a level to have platform that is easy to reach from multiple platforms.

```python
if ground_platform_count > max_ground:
    score += (ground_platform_count-max_ground)
if ground_platform_count < min_ground:
    score += (min_ground - ground_platform_count)
for platform in platforms:
    platform_count = traversed.count(platform)
    #every platform must be reachable
    if platform_count < 1:
        score += 1
```

Result

For Deep Space, the plot of an individual patrolling pattern follows what I originally thought the patterns should be like. However once in the game it became difficult to see that enemies followed a pattern. Adding multiple enemies together in a level made it almost impossible to see any patterns at all. The crosses and distance to goal parameters did work in regulating the difficulty but the difficulty came from the enemy movements seeming almost random.

Levels that are smaller in terms of size but also in terms of the number of enemies and patrolling points could create reasonable levels, like the level seen in figure 6. Larger levels that have more open space the resulting levels felt more unpredictable.

The level structure generated for Friday was more consistent at all sizes. The use of a simplified player physics model worked well in generating platforms that were reachable in the game. Since the path searching function searches for a way to climb along the y-axis, the levels get a feel of direction for the player to follow. This means that even though the platforms were in random sizes and almost random positions the levels did not feel random. In fact, I have made levels by hand before that feels more random since it was time consuming making sure all platforms were reachable or at least seemed to follow a path. Figure 7 shows how this kind of path looks like from some distance.

The variety in the generated levels consisted in different positioning and sizes of platforms but the overall feel were
mostly the same. It would be preferable to get some more visually distinct levels for a player to feel it is a different level. Changing some constraint variables can provide some visual difference as seen in Figure 8.

![Image](image_url)

**Figure 8.** A generated level for Friday with wider platforms.

The level generation in Friday found a good structure relatively fast but could then be stuck for a while correcting the position for a few bad platforms. The bad platforms are easy to notice and could often be fixed faster by manually editing the level than letting the GA complete. Since most platforms were still good, the reduced work for a designer is still significant. I added a very simple interface where a designer could choose to plot the best-found level. If this level is “good enough” a designer could stop the GA. This will cause the best-found level to be saved even though it does not meet all constraints. Figure 9 shows a level that is “good enough” and easy to fix manually.

![Image](image_url)

**Figure 9.** A generated level deemed to be “good enough”.

Creating constraints for a level in both games was challenging. Loose constraints could have interesting and new types of levels within the solution space but it will also introduce many levels that are not good. To solve this some researchers use two populations. For example in the paper “Towards a generic framework for automated video game level creation.”, Sorenso, Nathan and Philippe use a feasible-infeasable two population genetic algorithm to first solve the constraints and then search within the solutions for “fun” levels [17]. Instead of fun it is also possible to use other criteria’s as done in the paper “Enhancements to constrained novelty search: Two-population novelty search for generating game content.” by Liapis, Antonios, Georgios N. Yannakakis, and Julian Togelius. In this paper, a feasible-infeasable novelty search is described that searches for diversity within the solution space by looking at the distance between solutions [7]. In both cases, a second fitness function has to be produced, which is an added obstacle. A function that gives a score on how “fun” a level is or how to measure the distance between solutions is a non-obvious function. Another approach could be to limit the constraints creating a much tighter solution space. Creating constraints that limits the search to a more restrictive structure is easier to come up with in my experience. The constraints for Friday were limited around specific layout of platforms, in other games it could be the placement of resources or spawning position for enemies. However, a constraint that is easier to think of might not be easier to implement. The reachability constraint in Friday was solved with a search function that used a simplified physics model, which is arguable more complex than the implemented constraints for Deep Space.

I had not done any level at all for Deep Space when I was working on the fitness function. Instead, I used my idea of how a level should look like as a base to create the constraints. It would have been better to create a level by hand first to get a better feeling for the game and its limitations. This way I could have realized earlier that the enemy movement alone was difficult to recognize as patterns. Another problem with Deep Space was that the puzzle was time dependent. To use a player simulation as used with Friday would mean it would have to be simulated over time together with the enemy movements. Timing issues was also a concern in [10] where a story solution to the Hitman game could break due to unfortunate timing of the enemy movement.

**Performance**

In this thesis, I have not taken performance into much thought since I limited myself to smaller problems and designed it to be used in an offline setting. Since a direct representation was used, the time to find solutions grew exponentially with the size of the level. For smaller platform levels with about 20 platforms, it could take around one hour to find a solution using a eight core Intel® Core™ i7-3770 CPU running at 3.4 GHz. However, as soon as more platforms were added it could take days to complete. For the platform generation I did add some extra
optimizations by creating customized initialization and mutator functions that would only randomize values within the constraints of possible x, y and width parameters in order to reduce the search space and remove some constraints from the evaluation function.

**DISCUSSION AND FUTURE WORK**
The field of SBPCG is still young and evolving. In this thesis, I wanted to explore the use of SBPCG as a more general tool for level generation between different games. The approach I used with GA and constraints proved to be more difficult than I first thought to generalize. The sensitivity and importance of the encoding used puts a burden on the designer if this technique would be used directly. If an expert is used to setup the encoding and fitness function it is also a risk that the output would not be what the designer wanted. An added layer that can automatically create the encoding and fitness function needed to follow designers specification could be interesting research.

While it might not be suitable for a designer to work directly with a GA, it could be interesting to include a GA framework in game engines. The integration in game engines could help developers add generative tools to their level editors. It could also mean it would be easier to use simulation based fitness functions if the engine allows the game to run faster than real time.

**REFERENCES**


In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Jesper Lundgren