AUTOMATIC GAME SCRIPT GENERATION

By

Ahmed Abdel Samea Hassan Khalifa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
JUNE 2015

AUTOMATIC GAME SCRIPT GENERATION

By

Ahmed Abdel Samea Hassan Khalifa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Under the Supervision of

Prof. Magda Fayek
Professor of Artificial Intelligence
Computer Engineering Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
JUNE 2015

AUTOMATIC GAME SCRIPT GENERATION

By

Ahmed Abdel Samea Hassan Khalifa

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by the Examining Committee:

____
Prof. First S. Name, External Examiner

____
Prof. Second S. Name, Internal Examiner

____
Prof. Magda Fayek, Thesis Main Advisor

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
JUNE 2015

Engineer's Name: Ahmed Abdel Samea Hassan Khalifa
Date of Birth: 22/02/1989
Nationality: Egyptian
E-mail: amidos2002@hotmail.com
Phone: +2 0100 671 8789
Address: 20 Mahmoud El-Khayaal St, Haram, 12561 Giza
Registration Date: 1/10/2010
Awarding Date: 14/7/2015
Degree: Master of Science
Department: Computer Engineering

Insert photo here

Supervisors:

Prof. Magda Fayek

Examiners:

Prof. First S. Name	(External examiner)
Prof. Second S. Name	(Internal examiner)
Prof. Magda Fayek	(Thesis main advisor)

Title of Thesis:

Automatic Game Script Generation

Key Words:

Procedural Content Generation; General Video Game Playing; Genetic Algorithm; Computational Creativity; Rule Generation; Level Generation; Video Game Description Language; Puzzle Script; Puzzle Games

Summary:
Summary

Table of Contents

List of Tables

List of Figures

List of Algorithms

Acknowledgements

I would like to express my deepest gratitude to Micheal Cook. His wok on ANGELINA was my main inspiration to start working on Procedural Content Generation. I would like to thank my supervisor Prof. Magda Fayek for all the support, guidance, and extreme patience she provides. Without her support this work would not have saw the light. Also Thanks to all my friends for the huge help in collecting human feedback on the results.

Abstract

This file is provided to help graduate students at the Faculty of Engineering, Cairo University in preparing their theses according to the regulations and format guidelines defined by the graduate committee. Students are required to consult the regulations for thesis preparation available at the department of graduate studies besides using this template.

In this template, different styles are defined which start with âĂIJFECU ThesisâĂİ phrase. You may use these styles to quickly format your text throughout the thesis. You may also change these styles as long as they comply with the regulations for thesis preparation.

Chapter 1: Introduction

Video Games were originally created by small groups of people in their spare time. As time passed Video Games evolved to be a huge multi-billion industry where thousands of people are working everyday to create new games. Automation doesn't play huge role in creating games as most of the work is done by humans. Hiring humans ensures producing high quality games but it costs a lot of money and time.

1.1 Motivation

During the early days of Video Games, games were created by few people in their spare time. Most of the time was spent in programming the game, while a small portion was dedicated for graphics, sounds, and music because of the technical limitations of the devices at that time. As these limitation are no more, producing a game takes more time than before. Most of that time is spent on creating content for the game (graphics, music, sounds, levels, and ...etc)[66]; for example creating graphics for a huge main stream game may take hundreds of artist working for a year or two. That is why the production cost of a huge game reaches millions of dollars[24].

That huge production cost caused lots of Game Studios to shut their doors [16] and others became afraid of creating new game ideas. In spite of all these technological advancement, we could not reduce the cost and time for creating good game contents because creation process heavily depends on the creative aspect of the human designer. Automating this creative process is somehow difficult as there is no concrete criteria for judging creativity which raises the main question: Can technology help to reduce the time and money used in producing games?

1.2 Problem Statement

Creating content for games or any creative medium, e.g. art, music, movies, and ...etc, is really a tough problem and take a huge amount of time. For example a famous popular game called *Threes* took around 14 month working on the concept itself [60]. Game market is growing fierce as there is always a demand for new games with new contents but that can not be accomplished very fast because creating new innovative games requires long time.

Games consists of lots of aspects ranging from game graphics to game rules. It is hard to generate all these aspects at the same time, so we are going to address just two of them which are Levels and Rules. Level Generation has been in industry for decades since early days of games [1] but it has always used lots of hacks and it has never been generalized. Same applies to Rule Generation except that their is a very small contribution in automatically generating rules.

Complete automation of levels and rules for all types of games seems beyond the reach at the present time, so In this work we focus on automatic generation of Puzzle Games because they are accessible by all players, they do not require much time to be played, and do not have any boundaries on their ideas. From there, we found an urgent need to research for new ways in Level and Rule Generation for Puzzle Games.

1.3 Objectives

The aim of this work is to try to understand how our brain works in creating creative content, helping game designers and developers to think outside the box, and minimizing their time for searching for new content by providing them with good diverse seeds for levels and rules. It is not intended to create an Artificial Intelligence that can create the whole game from scratch and sell it afterwards because its too early to think that computers can do that on its own and people are not prepared yet to deal with that kind of Intelligence that can replace them at work.

1.4 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 explains the background needed to understand Procedural Content Generation in Video Games, conducted by Chapter 3 explaining any previous work done in that area. Chapter 4 shows different methodologies and techniques used to reach our objectives followed by our results from applying these techniques in Chapter 5. Finally, Chapter 6 presents our conclusion and future work.

Chapter 2: Background

This chapter presents some background information needed to understand this thesis. It starts with explaining Puzzle Genre in Video Games, followed by description of Procedural Content Generation and a special case called Search Based PCG. It is conducted by describing Video Game Description Language and why it is important. Finally, it explains General Video Game Playing and its relation to this research.

2.1 Puzzle Genere in Video Games

In 1950 a small group of academic people started developing Video Games on the main frames of their schools. This has continued to till the 1970s when Video Games reached Main Stream industry. In 1980s lots of technicals problems were solved which brought new types of Video Games. Game Genres were invented to differentiate between different game types so players can refer to their favorite games by genres, for example there is Adventure genre like Legend of Zelda, Fighting genre like Street Fighter, Platformer genre like Super Mario Bros, Shooter genre like Space Invaders, Puzzle Genre like Bomberman and ...etc. In 2000s people started to go toward being mobile and making games for cellular phones and one of the most important genres on these mobile devices is Puzzle Genre[23].

Puzzle Genre is a style of Video Games that emphasize solving puzzles, it always focuses on logical or conceptual challenges. Puzzle games can be described using the following aspects:

- **Graphics:** How the game looks like.
- **Inputs:** How input is given to the game.
- **Rules:** How the game behaves to events.
- **Levels:** How objects are placed.
- **Win Condition:** How to end the game.

2.2 Procedural Content Generation

Procedural Content Generation (PCG) means generating game content using a computer. It was first developed due to technical limitations (small disk space) and the need to provide a huge amount of content. *Akalbeth* was the first game using PCG, the game starts asking the player to provide it with a lucky number that is used as a seed to a random number generator which generates everything in the game from the player statistics to dungeon maps. Due to that the game has a very small footprint around 22 Kb so it does not need to be loaded from lots of disks [1]. Although technical difficulties become history and storage is no longer a problem, PCG is still one of the hot topics in Video Games Industry and Research. PCG helps us reduce development time and cost, be creative, and understand the process of creating game content. PCG can be used to generate different game aspects for example Textures, Sounds, Music, Levels, Rules, and ...etc. In this thesis we will focus only on the Levels and Rules Generation which is the core aspect for any Puzzle Game.

2.2.1 Level Generation

Levels are different combination of game objects. These combinations control the game flow and difficulty. For example in Super Mario, as the game advances the levels become longer and include more enemies and gaps. Level Generation has been in industry since dawn of games in order to decrease the game size, but it is now used to introduce huge amount of levels that humans can not generate manually in reasonable time. Level Generation has always been done for a specific game using lots of hacks to improve the output result. These hacks cause the output levels to follow certain guidelines which may cause elimination of huge amounts of possible levels but on the other hand these guidelines ensure that the output levels are all playable (can reach goal of the game) and satisfactory by all players [19].

2.2.2 Rule Generation

Rules are what governs game behavior. Based on them players can know how the game will be played; for example what to collect, what to avoid, how to kill enemies ...etc. Rules are very different from one game genre to another, for example in Board Games (e.g. Monopoly) all rules are applied after player plays his turn, while in Platformer Games (e.g. Super Mario) the rules are applied with every time step (not depending on the player movement). Due to that huge difference between rules in different genres, Rule Generation is one of the most difficult contents to generate. That is why we need to find a way to represent and describe rules. The only way to represent all different genres is using Coding Languages because Coding is the only common thing between all games. Generating new Rules using Coding Languages may take years because Coding Languages are used to create any computer related software such as plug-ins, programs, viruses, games and ...etc. Still some researchers managed to use it but it will need more guidance due to the huge search space of Coding Languages [9]. A better solution is to use a description language for the rules that is more specific for a certain genre which we are going to discuss in Section 2.4 [62].

2.3 Search Based PCG

There are many approaches to generate content in Video Games. The search based approach is one famous technique that has been used a lot in recent research. The search based approach can be divided into three main parts Search Algorithm, Content Representation, and Evaluation Function [63].

2.3.1 Search Algorithm

This is the *Engine* for generating game content. There are different searching algorithm for example Simulated Annealing, Hill Climbing, and Genetic Algorithm (GA) . GA is a search algorithm based on Darwin Theory *"Survival for the Fittest"*. GA uses techniques inspired from natural evolution such as selection, crossover and mutation in order to find solution to large space optimization problems [20].

2.3.2 Content Representation

There are several ways for modeling different aspect of the game content during the generation process. The way of representing the content can affects a lot on the output of the generation. For example levels may be represented as:[63]

- 2D matrix where each value represent a tile location in the level.
- Sequence of position of objects in level.
- Sequence of level Patterns.
- Level properties such as Number of Objects in map, Number of Players and ...etc.
- Random number seed.

2.3.3 Evaluation Function

Without Evaluation Function the Search Algorithm will be blind. The evaluation function leads the Search Algorithm to find better content in the solution space. The evaluation Function can be either one of the following:[63]

- **Direct evaluation function:** which utilizes some understanding about the generated content and evaluates it accordingly.
- **Simulation-based evaluation function:** Use Artificial Intelligence agent to test the generated content and based on his behavior it estimates its quality.
- **Interactive evaluation function:** Evaluate generated content based on the interaction with human.

2.4 Puzzle Script as Video Game Description Language

Referring to Section 2.2.2 we can not generate game rules without having a methodology to describe it. Video Game Description Language (VGDL) was originally invented to help on the work for General Video Game Playing (discussed later in Section 2.5) at Standford University which will be discussed in the following section. The idea behind VGDL was to provide a simple description language that can be used to describe simple 2D games. Researchers insist that any VGDL language must have the following aspects:[15]

- **Human Readable:** It must be easy for human user to understand games written with it and formulate new one.
- **Unambiguous and Easy to parse:** It must be easy to parse using a computer that is why it must be clear.
- **Searchable:** The games formulated using it can be drawn in form of tree so its easy to use Search Algorithm (discussed in Section 2.3.1) to find new games.
- **Expressive:** It can be used to express more than one game.
- **Extensible:** It can be extended to add more game types and dynamics.
- **Viable for Random Generation:** Each component of the language have 'sensible defaults' so its easier to search for new values for certain components without worrying about other ones.

Puzzle Script (PS) is a VGDL created by Stephan Lavelle to help game designers and developers to create puzzle games [38]. Games generated by PS are time stepped games similar to Sokoban game; Sokoban is an old Japanese puzzle game where your goal is to push some crates towards certain locations[51]. PS file starts with some meta data like game name, author name, and website then it is divided into 7 sections objects, legend, sounds, collision layers, rules, win conditions, and levels. Figure 2.2 shows an example of the PS file for Sokoban game. It is refereed to it in the next sections.

2.4.1 Objects

It is the first section of the PS file and it contains a list for all the object names and colors being used in the game for example in Figure 2.2 we had Background, Target, Wall, Player, and Crate associated with a color.

2.4.2 Legend

It is a table of one letter abbreviation and one or more group of objects for example in Figure 2.2 the Wall is assigned "#" symbol and Crate and Target together are assigned "@" symbol.

2.4.3 Sounds

It consists of group of game events associated with a certain number. PS engine use these numbers as parameter to BFXR to generate sounds in runtime[5] for example in Figure 2.2 Move event for Crate object is assigned to value 36772507.

2.4.4 Collision Layers

It defines which objects can not exists at same location during running the game, Objects on the same line can not exist together. Each line is considered a new layer, for example in Figure 2.2 Player, Crate, and Wall can not exists at same location as they are assigned to the same layer.

2.4.5 Rules

Its a set of production rules that govern how the game will be played. Production rules have the following formate

Tuple ... Tuple -> Tuple ... Tuple

where the number of tuples on the right must be equal to number of tuples on the left, and each tuple must be in the following format

[Objects | ... | Objects]

Each tuple has a group of Objects separated by "|" symbol which means that these objects are beside each other for example [Player | Crate] means there is a Player and Crate object beside each other. The number of "|" in both left and right side of each tuple must be equal. Objects must be written in the following format

Direction Object ... Direction Object

Object must be selected from the Objects section and must be from different collision layer for example [Player Target] means Player and Target exists on same tile. Last thing is the Direction, it must be one of the following ">", "<", "$\wedge$", and "$\vee$". The first one means in same as the moving direction, the second one means opposite to moving direction, while the third means 90° from the moving direction, and the final one is −90° from the moving direction. Let's take a real example on the rule, for example in Figure 2.2 the rule is

[> Player | Crate] -> [> Player | > Crate]

means if there is a Player and Crate beside each other, and the Player moves towards the Crate, then both the Player and the Crate will move in the same direction.

2.4.6 Win Conditions

It identifies when the level should end. It consists of 2 objects separated by on keyword. For example in Figure 2.2 "all Target on Crate" means that the level ends when all Targets are on same location with Crates. The possible keywords that can be used are "all", "some", and "no". "all" means every single object must be at same location, while "some" means at least one object at same location, but "no" means that the two objects must not be at same location.

2.4.7 Levels

They are 2D matrices showing the current configuration for each game level using the abbreviations found in the Legends section. Each level is separated from the next level by a new line. In Figure 2.2 you can see 3 designed Sokoban levels. By replacing each symbol by the corresponding object from the Legend section then the color from Objects section, so you will get the following levels in Figure 2.1.

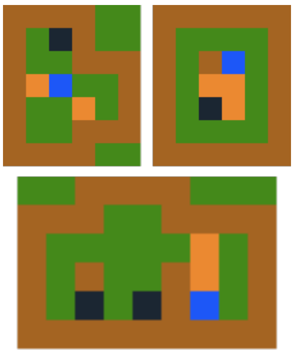

Figure 2.1: Puzzle Script Levels

```
title My Game
author Stephen Lavelle
homepage www.puzzlescript.net

========
OBJECTS
========

Background
GREEN

Target
DarkBlue

Wall
BROWN

Player
Blue

Crate
Orange

=======
LEGEND
=======

. = Background
# = Wall
P = Player
* = Crate
@ = Crate and Target
O = Target

 =======
 SOUNDS
 =======

 Crate MOVE 36772507
 endlevel 83744503
 startgame 92244503

================
COLLISIONLAYERS
================

Background
Target
Player, Wall, Crate

======
RULES
======

[ >  Player | Crate ] -> [  >  Player | > Crate  ]


==============
WINCONDITIONS
==============

All Target on Crate


=======
LEVELS
=======

####..
#.O#..
#..###
#@P..#
#..*.#
#..###
####..

######
#....#
#.#P.#
#.*@.#
#.O@.#
#....#
######

..####...
###..####
#.....*.#
#.#..#*.#
#.O.O#P.#
#########

```

Figure 2.2: Puzzle Script File

2.5 General Video Game Playing

General Video Game Playing (GVGP) is an area in Artifical Intelligence (AI) where people try to develop an intelligent agent that is able to play more than one game. Most of Game Playing Agents are not general players, they are designed for a certain game like Chess, Poker, Backgammon, or ...etc. At the beginning the AI is supported with game rules in form of VGDL Section 2.4 with current game state and possible actions and the agent should choose the next action. After several play-outs the agent should learn how to maximize his score or reach goal and avoid death. Researchers used different methods to try to create general AI using Monte Carlo Tree Search, Neural Networks, and Reinforcement Learning methods. GVGP is very important for PCG specially for rule and level creation because in order to generate good random level or rule, it is better to test it out using a GVGP Agent to have a better judgment on how the random content will be played.[27]

Chapter 3: Literature Review

This chapter will provide a review of the past work on Procedural Content Generation. It will highlight different efforts towards generating levels and rules for games.

3.1 Level Generation

This section will present some of the work related to Level Generation. Most of the previous work did not talk a lot about generating levels for Puzzle Games. It focused more on Platformer and Arcade Games. The work found about Puzzle Games was limited toward specific games such as Sokoban[57, 31], Cut The Rope[46], and Fruit Dating[37]. Even the generic work on Puzzle Games needs prior human understanding of the game rules[43, 42].

Generating levels for Puzzle Games seems interesting but it has lots of problems. Some of these problems are the main reason for not having enough previous work.

- All levels must be solvable (have at least one solution).
- Most of the level objects should be used (adding unused items may be considered as a bad level design).
- Puzzle Games ideas have no restriction. For example some games have continuous space where game actions depend on player's skills and perfect timing (Angry Birds[2], Cut the Rope[10], and Where's my Water?[67]) while other games have discrete space where game actions depend on the best sequence of actions regardless of the time taken (Fruit Dating[18], Sokoban[51], and HueBrix[25]).

3.1.1 Puzzle Level Generation

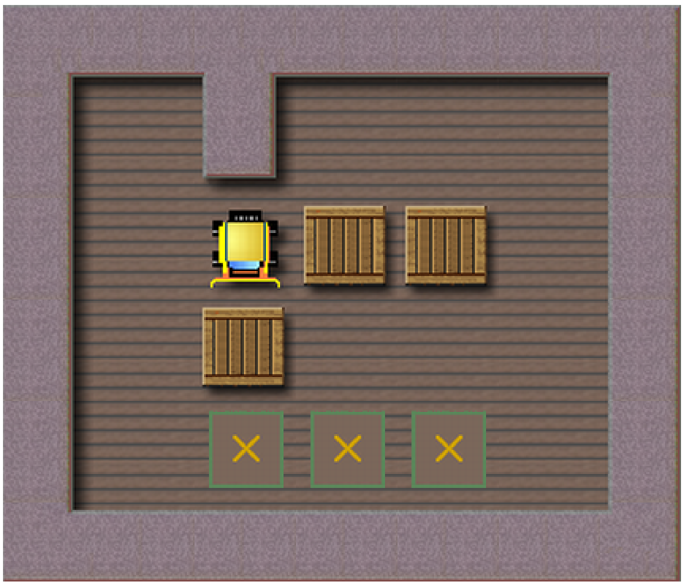

Figure 3.1: Screenshot from Sokoban

As there is nothing before like this work, this section will show all previous work that can be slightly related to our problem. One of the earliest research in Puzzle Games was by Murase et al.[31]. Murase et al. work focused on generating well designed solvable levels for Sokoban[51]. Sokoban is an old Japanese puzzle game where your goal is to push some crates towards certain locations as shown in Figure 3.1. Their work consists of 3 stages:

- **Generate:** Responsible for generating Level Layouts. Level Layouts is generated by placing predefined templates at random positions. Goal locations and crates are placed afterwards at random positions such that each goal location is reachable from the player location and each crate is reachable from a unique goal location.
- **Check:** Responsible for checking for playability. The system uses Breadth First Search (BFS) to check for solution. All unsolvable levels are removed in this step.
- **Evaluate:** Responsible for selecting best generated levels. The system removes all levels that have a solution that is very short, contains less than four direction changes, or does not contain any detours.

Results show that for every 500 generated levels only 14 are considered as good levels. These good levels are characterized by having a short solution sequence due to the usage of BFS.

Taylor and Parberry[57] followed Yoshio Murase et al.[31] work to improve generated level quality. Their system consists of 4 stages:

- **Generating empty room:** Responsible in generating Level Layouts like Yoshio Murase et al.[31] work. After generating the layouts, the system discards any level that is not completely connected, has huge empty space, or has number of empty floors less than the planned number of boxes and goals.
- **Placing goals:** Responsible for finding best suitable goal locations. The system use a brute force algorithm to try every possible combination for goal locations.
- **Finding the farthest state:** Responsible for placing crates at farthest location from its goal location. This process is done using a similar algorithm to BFS. The algorithm expands all the reachable locations from a goal location and returns the farthest location. The farthest location is calculated using a box line metric heuristic which calculates the number of unrepeated moves required to reach that location.
- **Evaluating generated levels:** Responsible for evaluating the generated levels. Evaluation is done using some heuristic functions. For example: number of box pushes in the solution, number of levels found at same solution depth, number of box lines, number of boxes, and ...etc.

The generated levels are not suffering from the problem of short level sequences presented in Yoshio Murase et al.[31] work. As the target number of crates increases, the generation process takes more time but delivers much more interesting and difficult levels.

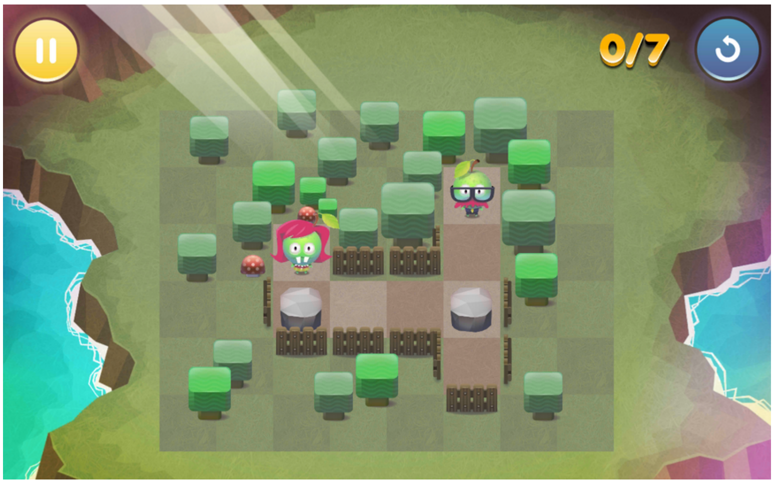

Figure 3.2: Screenshot from Fruit Dating

Rychnovsky[37] work focused on generating levels for his new game Fruit Dating. As shown in Figure 3.2, Fruit Dating is a puzzle game where the player need to move all similar fruits beside each other. The game is played by swiping in one of the four direction (up, down, left, right). Swiping in any direction causes all objects to move towards that direction. Rychnovsky developed a level editor that can be used to generate new levels or test playability of certain level. Generating new levels is done using the following steps:

- **Generating level structure:** The system generates level structures based on 2 steps. First step is generating external walls. External walls are generated at any random location connected to the border with short random length. Second step is generating inner walls. Inner walls are generated at any free location with free 3x3 locations.
- **Placing game objects:** The system generates fruits at random locations based on predefined weights. Every empty location is assigned a score using these weights. Locations with highest scores are selected. Other game objects use the same strategy but using different weights.
- **Checking for a solution:** The system use an algorithm similar to BFS to find the solution of the generated level. If no solution found, the level is discarded.

The technique doesn't take more than couple of minutes to generate a level. The main problem is there is no way to influence difficulty in the generated levels.

Figure 3.3: Screenshot from Cut The Rope

Shaker et al.[42] worked on generating levels for physics based puzzle games. They applied their technique on Cut The Rope (CTR) . As shown in Figure 3.3, CTR goal is to cut some ropes to free the attached candy to reach OmNom (a frog monster fixed at certain position on the screen). As game progress more game objects are introduced. These objects help in changing the movement direction of the candy to redirect it towards OmNom. Shaker et al. used Grammar Evolution (GE) technique to generate levels for CTR. GE is a result of combining an evolutionary algorithm like GA with a grammar representation. The grammar is designed to ensure that every game object appears at least one time. The fitness function used to rate the generated levels depends on some heuristic measures based on prior knowledge about the game. For example: Candy should be placed higher than the OmNom, OmNom must be placed below the lowest rope, and ...etc. Since using heuristics does not ensure playability, each generated level is played 10 time using a random player. The fitness score is reduced by a large number if no solution was found. Shaker et al. generated 100 playable levels and analyzed them according to some metrics such as frequency, density, and ...etc.

Shaker et al.[46] conducted their research on CTR to improve generated level quality. They replaced the random player with an intelligent one. Two types of automated players are developed. The first one takes an action based on the current properties of all objects in the level. While the second one takes an action based on the reachability between every level objects and the candy at every time step based on the candy's current position, velocity, and direction. The generated levels are far more diverse because the random player discards some potential levels in the search space.

Shaker et al.[43] introduced a new generation technique named Progressive Approach. Progressive Approach can be used on any kind of games to reduce the generation time. Progressive Approach is divided into two main steps:

- **Time-line Generation:** GE is used to evolve a time-line of game events.
- **Time-line Simulation:** Evolved time-lines are simulated using an intelligent agent. The agent utilize prior knowledge about the game to map the time-line to a possible level layout. Based on the agent result and some desirable heuristics, each time-line is assigned a fitness score.

Shaker et al. tested the new technique on CTR and compared its results with their previous work[46]. The results indicates a huge decrease in generation time, as it changed from 82 seconds towards 9.79 seconds. Although Progressive Approach is much faster and better from previous, it is difficult to determine the level difficulty before simulation. Also the quality of the levels depends on the intelligent agent used in the mapping process.

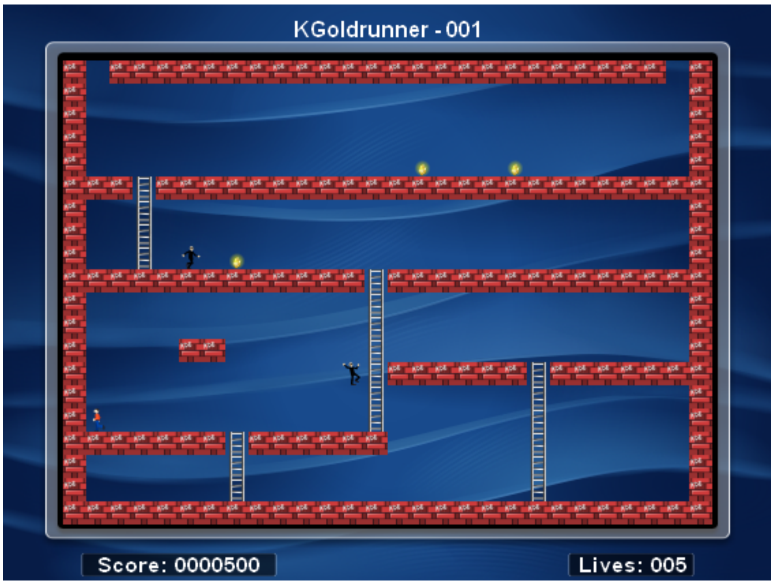

Figure 3.4: Screenshot from KGoldRunner

Williams-King et al.[68] work focused on generating levels for their game KGoldRunner. KGoldRunner is just a port for the famous game Lode Runner[30]. As shown in Figure 3.4, the goal is to collect all gold chunks without getting killed. Player can move left, move right, climb a ladder, move across a bar, or dig the floor. The game contains enemies who changes the position of the gold chunks. After collecting all gold chunks a hidden ladder appear which lead the player to the end of the level. Williams-King et al. used GA to generate random levels. Generated levels are evaluated using fitness function based on some prior knowledge about the game. For example: checking level connectivity using BFS between starting point and all gold coins. High score levels are simulated using an automated player that tries to collect all gold chunks using 20 different paths. The number of solvable paths indicates the level difficulty. As the number of generations increase more time is required but more levels are likely to be solvable.

Figure 3.5: Screenshot from Refraction

Smith et al.[47] worked on generating puzzle levels for Refraction. Refraction is an educational game created by Center for Games Science at the University of Washington. Refraction teaches children concepts in math through a puzzle game. As shown in Figure 3.5, the goal is to make every alien spaceship fully powered by directing a laser beam towards them. Each ship needs a fraction of the laser to operate. Player should use tools like Benders (change the direction of laser by 90°), Splitters (divide laser power by number of outputs from it), and ...etc to achieve that goal. Smith et al. divided his generator into 3 main components:

- **Mission Generation:** Responsible for generating a general outline showing level solutions.
- **Level Embedder:** Responsible for translating the general outline to a geometric layout.
- **Puzzle Solver:** Responsible for testing generated level for solution.

These components are implemented into two different ways (Algorithmic approach and Answer Set Programming (ASP)). Results shows that ASP is faster than Algorithmic approach specially in Puzzle Solver module, while Algorithmic approach produces more diverse levels than ASP.

3.1.2 Other Level Generation

This section is going to focus on Level Generation for non Puzzle Games. The previous work in that field focused on different game genres but a lot of work was done in Platformer genre. Showing some of the work will help in understanding the current research direction towards Level Generation.

Constructive approach in PCG has been used in commercial games for long time. It had been used in generating textures, trees, sound effects, and levels. Ismail and Nijman[39] used an Agent Based PCG to generate levels for their commercial game Nuclear Throne.

The system spawns an agent at a random position on the map. This agent starts moving into a random direction with a certain probability to change direction. New agents are spawned based on a certain probability. The agents continue moving till the number of dug floors reached 110 or reaching a dead end. Coxon[33] used perlin noise and cellular automata algorithm to generate a huge map for his commercial game Lenna's Inception. The game map is divided into small rooms. Perlin noise is used to assign a certain terrain type for each room. Cellular automata algorithm is used to craft the details for each room while ensuring reachability between entrances and exits.

Search algorithms (specially GA) has been used a lot in level generation specially in academia. Sorenson and Pasquier[53] worked on improving GA to adapt with level generation for any game. The new technique divides the population into Feasible and Infeasible. Each population is evolved on its own using crossover and mutation. Chromosomes can transfer from Feasible to Infeasible population and vice versa. The Feasible population uses a fitness function that measures the challenge presented in the game. The Infeasible population uses a fitness function that measures the distance towards the Feasible population. These techniques were tested over two different games The Legend of Zelda and Super Mario Bros. The results shows promising levels generated for both games which is an indication for the possibility of using this technique as a generic framework.

Preuss et al.[36] used 3 different search techniques to generate diverse game maps for Strategy Games. They used Restart Evolution Strategies, Novelty Search, and Niching Evolutionary Algorithm then compared their results in the end. The fitness function used in evaluation consists of 3 metrics:

- **Tile Based Distance:** measures diversity between each two maps.
- **Objective-based Distance:** measures the quality and playability of the map according to some heuristics for Strategy Games.
- **Visual Impression Distance:** measures how good the map look using 20 different heuristic measures.

Niching Evolutionary Algorithm generated the bested results in diversity and quality measurement, followed by Restart Evolution Strategies which is almost the same, while Novelty Search was the worst of them in both quality and diversity.

Liapis et al.[28] improved Novelty Search Algorithm to perform better than previous work[36]. The improved algorithm borrowed an idea from Sorenson and Pasquier work[53] by dividing the population into Feasible, Infeasible populations. The Feasible population uses a fitness function that measures the distance between chromosomes to ensure diversity. The Infeasible population was tested using two different fitness functions. The first one is same like the feasible population, while the second one uses the distance toward feasibility like Sorenson and Pasquier work[53]. The results prove that the second fitness generates more feasible solutions but less diverse compared to the first one.

Baghdadi et al.[3] used GA to evolve levels for the commercial game Spelunky[54]. Spelunky entire map is divided into 4x4 rooms. Levels are represented as a group of integers. These integers represents positions for start room, exit room, room connections, number of enemies, and ...etc. Each level is mapped to a layout before evaluation. Mapping process consists of 3 main steps:

- Generating internal structure for each room using an Agent Based PCG.
- Adding some corridors to ensure connection between rooms.
- Distributing monsters, ladders, and bombs equally across the level.

Levels are evaluated using a fitness function consisting of two heuristics measures. The first heuristic ensures level playability by measuring connectivity between mandatory rooms, placement of starting room, and ...etc. Second heuristic measures level difficulty by measuring path length, number of enemies, and ...etc. The results shows that the introduced technique is able to generate a good designed levels for Spelunky with a certain difficulty.

Smith et al.[49] generated levels for Super Mario Bros (SMB) using Rhythms. Rhythms are a sequence of action user during SMB to finish the current level. Rhythms is a way to express the pace of playing a level. The generator is divided into 2 main parts:

- **Rhythm Generation:** Responsible for generating a sequence of actions required to finish the level.
- **Rhythm to Geometry Generation:** Responsible for translating the generated Rhythm into a corresponding level layout.

The generated level is tested against some critic measures provided by the human designer. These critic measures helps in enhancing the quality and ensuring the playability of the generated levels. The results shows the importance of including the pace of playing in generating platformer levels.

Sorenson and Pasquier[52] used GA to generate SMB levels based on previous studies about GameFlow[56, 7, 55]. GameFlow studies measure the player enjoyment as the amount of challenge facing the player. The studies recommend keeping the challenge level at an optimal rate (too much challenge cause frustration, too little challenge cause boredom). Based on that fact, The study modeled the challenge level as the player's Anxiety. Sorenson and Pasquier used anxiety curve as a fitness function to evaluate the generated levels. The rate of change of anxiety curve is modeled using the following equation:

$$\frac{df}{dt} = m * \frac{da}{dt}$$

Where m can be +1 or -1. The value of m is +ve all the time till a certain threshold. If this threshold is exceeded, m value become -ve causing the curve to decrease till reach another threshold where it starts increasing again. They compared their results with the original SMB levels. The anxiety curved of the generated levels are not similar to anxiety curve of the original SMB where anxiety curve drops instantly after reaching high value. This analysis proves that anxiety curves show a promising direction of using GameFlow as a measurement.

Shaker er al.[45] used GE to generate levels for SMB. The level is represented using context free grammar that represents the level as a group of chunks where each chunk can be any game object. GE tries to increase the number of chunks appearing in the level while minimizing the number of conflicts between them. The results of the generator is compared with two other generators. The results shows that level characteristics depends on the generation technique.

Some commercial games generate levels by combining a group of hand crafted game chunks to ensure the high quality of the level. Yu and Hull[59] divided Spelunky map into 4x4 rooms. Each room is chosen from a set of predefined rooms ensuring the existence of a path between start and end points. The generator modifies each selected room by adding some new blocks without blocking any movement path. Enemies and items are added after selecting the layout. Also Mcmillen[58] used a similar technique for his commercial game *The Binding of Isaac*. Due to the huge success of these games, that technique grows popular in academia.

Dahlskog and Togelius[11, 12, 14, 13] published several papers on generating levels for SMB game utilizing the patterns found in the original game. They started[11] by analyzing all original levels from SMB and detecting all repeated patterns. Repeated patterns are divided into 5 main categories (Enemies, Gaps, Valleys, Multiple Paths, and Stairs). Each category is divided into multiple configurations called Meso Patterns, for example Enemies pattern can appear as an Enemy (single enemy), 2-Horde (two enemies together), 3-Horde (three enemies together), 4-Horde (four enemies together), and a Roof (Enemies underneath hanging platform). The level generator generates levels by randomly choosing a group of these patterns and generating a geometry corresponding to them.Levels generated by that technique have the same feeling like original SMB levels. They conducted their research[12, 14] to improve the quality of the generated levels by using these analyzed patterns as an evaluation function. Levels are generated from concatenating a group of vertical slices called Micro Patterns. Micro Patterns are generated from original levels in SMB. GA is used to improve the generated levels using a fitness function that reward levels which contains more Meso Patterns. The generated levels are more similar to the original levels of SMB and more diverse. They conducted their research[13] to improve the quality of the generation by introducing Macro Patterns. Macro Patterns are a group of Meso Patterns after each other which can be used to identify the level of difficulty. SMB levels were analyzed and Macro Patterns are extracted from it. Macro Patterns from SMB is used as fitness function for GA. The results takes more time to generate but they have the same flow of SMB levels.

Shaker and Abou-Zleikha[44] used Non-Negative Matrix Factorization (NMF) technique to capture the patterns of level design of SMB. NMF can be used on any game if it has a huge amount of levels to learn the pattern from them. They used 5 different PCG techniques to generate 5000 levels for SMB (1000 level from each technique). NMF method factorize these levels into two new matrices:

- **Pattern Matrix:** captures the patterns found in each vertical slice of SMB.
- **Weight Matrix:** captures the percentage of each pattern to appear in the current level.

Pattern Matrix can be used to generate same levels like any of the input generators if its original weight vector is used. Shaker and Abou-Zleikha generated levels using free weights and compared them with all the results from the five generators. The results shows that NMF explores more of the search space than all the five generators, but They don't know if the whole space provides a good player experience.

Snodgrass and Ontanon[50] used Markov Chain Model to generate levels for SMB. Markov Chain Model is another way to learn about the design patterns from original SMB levels. The technique can be used on different game genres if there is a huge amount of highly designed levels. They tested the technique with different input parameters and evaluated the generated levels using human players. The results of the best input configuration only generate 44% playable levels.

3.2 Rule Generation

This section will show the latest research and the different techniques used to generate rules for different game genres. Most of the work in this section target generating Arcade games[61, 8, 40, 48, 64]. Arcade games are the most popular in research because they are easier to generate (simple rules), they provide a huge diversity in mechanics, and there is a huge research happening in GVGP for Arcade Games[27]. Other work tries to generate Board Games[6], Card Games[17], and Platformer Games[9] but as far as we know no one tried to generate Puzzle Games except for the work by Lim and Harrell[29].

3.2.1 Puzzle Rule Generation

Puzzle rule generation was infamous till 2014., when Lim and Harrel[29] published their work on generating new rules for PuzzleScript games. They use GA to find other possible rules that can solve a certain level for a certain game. For example what are the rules that can solve Sokoban level shown in Figure 3.6? Lim and Harrel used GA to generate rules. They divided the fitness function into 3 parts:

- **Rule Heuristics:** measures some logical constrains that should be found in rules. For example player should be on the left hand side of at least one rule, all movements in a certain rule should be in the same direction, and ...etc.
- **Validity:** checks for runtime errors in the generated rules.
- **Feasibility:** checks if the level is solvable using an automated player.

After running GA for 50 generations on the level shown in Figure 3.6, the system discovered new rules such as Crate Pulling, Morphing, and Spawning. The discovered rules are not new for human designer but they show promising direction in generating rules for Puzzle Games.

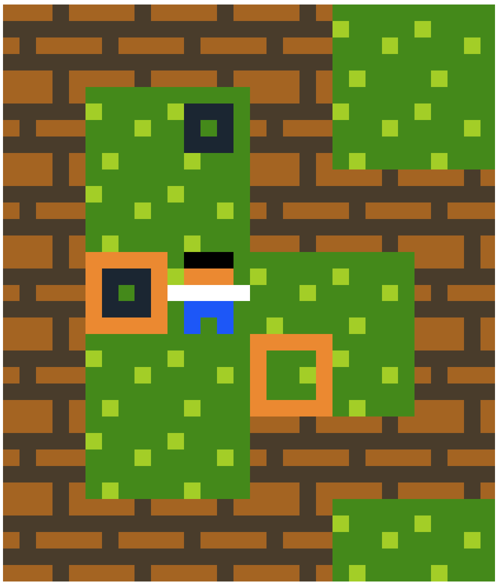

Figure 3.6: First level of Sokoban

3.2.2 Other Rule Generation

Although Rule Generation is the hardest content to be generated, a computer software written by Browne and Marie invented a board game called Yavalath. Yavalath is listed in the top 100 abstract board games of all time[69]. Browne and Maire software[6] used standard evolutionary approach to evolve game rules written in GDL for Combinatorial Games. Combinatorial Games are games that have finite number of states, deterministic (no randomness), discrete (one step at a time), perfect information (everything is visible), and involve two or more players. Evolved games fitness were measured by automated players for several plays (playouts). Based on the results of the playouts a score is given for each game based on 57 different heuristic metrics. The best scored games were tested by human players where each tester assign a score for these games. Comparing human scores with the system scores shows a correlation between both of them.

Jose et al.[17] used Genetic Programming (GP) to generate rules for card games using GDL specified for card games. The generated games are tested using automated players for several playouts. Based on the results of these playouts, games are assigned a fitness score. The system removes all games that never been finished in at least one of the playouts, results in draw for more than 20% of the playouts, or have more than 10 stages for one round. Other games are evaluated based on the difference between the number of wins of the best player to the worst player, the average number of turns required to finish the game, and the average number of turns with no actions. The resulted games are well formed and can be played by humans but at the same time they are boring and not interesting as most of them can be won using simple strategy.

Togelius and Schmidhuber[61] worked on generating rules for simple arcade games. Games are encoded in the form of collision and score matrices and a list of object behaviors. The collision matrix shows the result of colliding any two game objects. For example *red, white, kill, none* means If a red object collides with a white object, the red object will be killed while nothing will happen to the white object. Score matrix is defined in the same way but with values of +1, 0, -1. The list of object behaviors shows how different objects behave in the game (random movement, clockwise movement, still, and ...etc). All generated games terminate when $score \geq score_{max}$ or when $time \geq time_{max}$. Togelius and Schimdhuber used hillclimbing algorithm to generate game rules. Each iteration new game is generated and evaluated using a fitness function. The current game will be replaced with the new game if the new game scores better. The fitness function is based on the idea of learning progress which is inspired by Koster's Theory of Fun[26] and Schmidhuber's theory of curiosity[41]. A game is considered as a good game according to how much it can be learned. Based on that idea Evolutionary Strategies algorithm is used to evolve a neural network to play the generated games. Neural Networks output is used as a fitness score. That approach needs very long time to run that is why the best generated games are just playable games but not interesting.

Cook and Colton[8] took a new approach in generating full Arcade Games. They divided the game into 3 main elements:

- **Map:** is a 2D matrix showing passable and impassable tiles in the levels.
- **Layout:** is a list of position for every object on the map.
- **Rules:** govern how game works and they are represented in the same way in Togelius and Schmidhuber work[61].

Each game element is evolved using GA to maximize its own fitness score. For example map fitness measures the reachability of each tile, layout fitness checks for the distance between different objects, and rules fitness checks for game playability. All game elements communicate with each other through central game object. After each generation every game element updates the central game object with the best scored output. The system is very easy to add/remove game elements but some game elements can finish evolution early which minimize communications between different elements. Most of the generated games are not interesting although some of them have some similarity with the famous game PacMan.

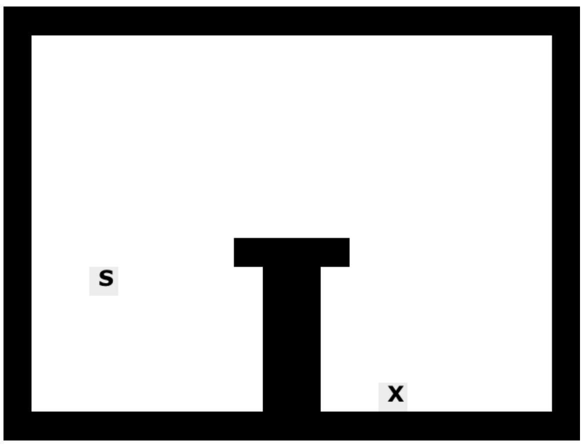

Figure 3.7: The challenge level for the Toggleable function

All discussed techniques so far based on having a specific GDL that governs the generation of rules. Cook et al.[9] decided to generate new game rule for his platformer game (A Puzzling Present) by analyzing and generating actual game code. They use code reflection ability to get all data members from the player class. Cook et al. used GP to evolve a new Toggleable function that helps an automated player to overcome a challenge level. A Toggleable function is any function where its effect can be reversed. Toggleable function is applied to one of the player data members. Cook et al. used the challenge level presented in Figure 3.7 to test the Toggleable function. The challenge level is designed to ensure no possible solution using platformer rules (moving and jumping). After the algorithm finds a Toggleable function, the system starts generating levels using GA. Levels are rated based on 3 simulation run to ensure certain characteristics.

- **First Simulation:** Check for a solution without using the new Toggleable function.
- **Second Simulation:** Check for a solution using the new Toggleable function.
- **Third Simulation:** Check for level difficulty based on the solution length.

Some of the best evolved Toggleable functions are Gravity Inversion, Teleportation, and Bouncing. The mechanics and levels are rated using human users. Human users rated Gravity Inversion and Bouncing higher than Teleportation as they are more understandable.

Farbs released a game called *ROM CHECK FAIL*[40]. ROM CHECK FAIL is a game which changes the rules applied on each object every few seconds. The new rules are selected from a pool of handcrafted rules. Some of rules controls player movements, enemy movements, object collisions, or ...etc. For example the player can be Link from The Legend of Zelda where he can move in all direction or a Spaceship from Space Invaders where he can only move left and right and shoot upwards. Although the game seems unfamiliar and weird, it has made a huge impact on the gaming scene.

Smith and Mateas[48] went for another direction in generating new games. Instead of generating a game then test it against set of constrains and/or evaluation function. They decided to limit the generative space for not exploring these parts to minimize the processing time. They used ASP to generate games with required aspects and constraining the unplayable parts in the generative space. For example putting a constraint for only winning if all white objects are killed, Ensures ASP not exploring other games having a different winning condition. Smith and Mateas tested their theory with Arcade games by creating a game called Variation Forever[65] which is inspired by ROM CHECK FAIL[40]. Games are described using the same way like Togelius and Schmidhuber[61]. Results show promising direction in using ASP for limiting the generative space.

3.3 General Video Game Playing

This section presents the latest work in GVGP either as standalone research or as a part of level or rule generation algorithms. Most of the discussed work with level generation either use BFS algorithm[31] or a tailored AI specially for a certain game[46]. For most of Level Generation work a tailored AI performs better than using a general one as tailored AI is designed with previous knowledge about the game. Utilizing this knowledge helps the automated player to find the solution quickly.

That is not the case with Rule Generation. Rule Generation searches for new unseen games where automated player is used to test its playability. Tailored AI will not work as there is no prior knowledge. Most of work done in Rule Generation used some kind of Search Algorithm (variants of BFS) to find the solution except for Togelius and Schmidhuber[61] work. They used an evolutionary Neural Network to measure learning progress through the evolved game. Browne and Maire[6] used Min-Max trees with Alpha Beta Pruning for automating game plays. They utilize the current game description to provide estimates for each board configuration. Jose et al.[17] used two types of automated players. The first one is random player which chooses random actions to model weak players, while the second is based on Monte Carlo Tree Search (MCTS) algorithm to model professional player. Cook et al.[9] used a normal BFS technique to search for a result for the challenge level. Lim and Harrel[29] compared BFS algorithm to solve generated games with Best First Search (BestFS) algorithm. BestFS is similar to A* algorithm where it sorts the explored nodes based on a heuristic function. The system utilize the knowledge about the goal condition and tries to minimize the distance between goal objects. The system also minimizes the distance between the player object and goal objects based on the fact that the player is the main object affecting the game world. Results show that BestFS finds solution faster than BFS but with slightly longer moves.

Most of GVGP research was done on Arcade games by ATARI 2600. One of the first work on that direction was the work done by Bellemare et al.[4]. They developed a system called Arcade Learning Enviroment (ALE) which contains lots of different arcade games from ATARI 2600. They tested their system against two different AI approaches:

- **Learning techniques:** Reinforcement learning algorithm called $Sarsa(\lambda)$ is tested using different types of features. Features ranged from using screen pixel colors to using the 1024 bit of Atari RAM.
- **Planning techniques:** Normal BFS and BFS with Upper Confidence Bounds Applied to Trees (UCT) are tested.

Results show that both approaches performed better than the average normal player. Although Planning techniques outperformed Learning techniques, it took more time to decide each action.

Hausknecht et al.[22] evolved a NN to play ATARI 2600 games. The system consists of 3 main steps

- **Visual Processing:** The system processes the game screen and detect all moving objects and group them into classes based on velocity information.
- **Self-Identification:** The system identifies which class is the player. The player object is identified by calculating the entropy over velocities for each class. The class with the highest entropy is considered the player.
- **Perform Action:** The system chooses its next action based on the positions of every object.

Their system outperformed three other reinforcement learning techniques based on their performance in playing two Atari 2600 games (Freeway and Asterix).

Perez et al.[34, 35] summarized the techniques and the results from General Video Game AI Competition (GVG-AI) [21]. GVG-AI is a competition that takes place every year for creating a General Player that can play some unseen ATARI 2600 games. Lots of techniques and methods used in the entries for the competition. Some of the techniques are based on learning methods while others on general heuristics. Heuristic methods produced better results than most of learning methods in the competition. The best algorithm is an Open Loop Expectimax Tree Search, followed by an algorithm named JinJerry Algorithm which is a variant of MCTS, then some variants of BFS. The first learning algorithm to appear in table of results is at the sixth place. The learning algorithm is a reinforcement learning technique called Q-Learning Algorithm which models the world in a form of Markov Decision Process.

Nielsen et al.[32] tested different AI techniques over ATARI 2600 games. They used different AI technique varies from MCTS to DoNothing. AI techniques are tested against 20 handcrafted games, 200 mutated version, and 80 random games. Based on the results, most intelligent techniques performed better on handcrafted games than other games. On the other hand, the DoNothing algorithm perform very bad on handcrafted games than other games. These results strengthen the idea of the need of an intelligent player to judge procedural generated games.

Chapter 4: Methodology

This chapter describes our approach for generating Puzzle Script levels and rules using different methodologies. This chapter is divided into two parts Level Generation and Rule Generation.

4.1 Level Generation

Level Generation is not an easy task specially when the game rules are not known before generation. Most of the previous work in the Puzzle Level Generation (refer to Chapter 3) was limited for generating levels for a specific game. Although some research suggested a general technique to generate levels, it is still based on designing a game specific fitness function. In this work, we suggest some global metrics for Puzzle Games that can help in generating levels with the minimum prior knowledge.

Our approach relies heavily on the understanding of the current game rules and some prior knowledge about Puzzle Script language. Figure 4.1 shows a high level block diagram of the system. The following subsections will describe each block in details.

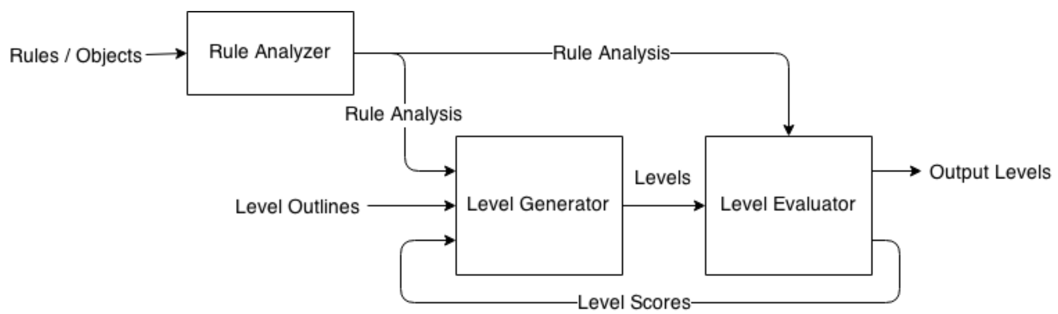

Figure 4.1: High level system block diagram for Level Generation

The system starts by analyzing the current game rules using a Rule Analyzer. The Rule Analyzer utilizes some basic information about Puzzle Script rules to understand the importance of each game object and its basic functionality.

The output of the Rule Analyzer (Rule Analysis) and the Level Outlines are fed to a Level Generator. The Level Generator is responsible for generating initial level layouts. It utilizes the Rule Analysis to insert game objects at suitable positions in the Level Outlines. Level Generator uses two different approaches: a Constructive Approach and a Genetic Approach. The Constructive Approach is faster in generation but produce less diverse levels, while Genetic Approach requires more time but give access to a vast majority of levels.

The generated levels are subjected to a Level Evaluator. The Level Evaluator uses an automated player to play the generated levels. Based on the result of each play, the Level Evaluator gives a score for the level based on some heuristic measures. These measures make sure the resulting level is playable and not trivial.

In case of the Constructive Approach, the system selects the best scored levels to output them, while the Genetic Approach enhances the output levels using GA.

4.1.1 Rule Analyzer

The Rule Analyzer is the first module in our system. It analyzes game rules and extract some useful information about each object. The extracted information is fed to the Level Generator and the Level Evaluator. Each object is assigned:

- **Type:** Object type depends on its presence in the Puzzle Script file. There are 4 different types:
  - **Rule Object:** Any object that appears in a rule is defined as a rule object. Rule objects are essential for rules to be applied.
  - **Player Object:** It is defined by name "Player" in the Puzzle Script. It is the main game entity. It can move freely without any restriction. Any level must have at least 1 player object. Player object is a Rule Object as well.
  - **Winning Object:** They are objects appearing in the winning condition. At least one of them is a Rule Object or a Player Object.
  - **Solid Object:** All objects that does not appear in any rule but on the same collision layer with a Rule Object.
- **Subtype:** each Rule Object is assigned a Subtype based on its presence in game rules. These subtypes are:
  - **Critical Object:** is an object that has appeared with the Player object and one of the Winning Objects in the rules.
  - **Normal Object:** same like the Critical Object but it only appears with one of them.
  - **Useless Object:** is an object that neither appears with the Player Object nor the Winning Objects in any rule.
- **Priority:** It reflects the number of times each object appears in the rules.
- **Behaviors:** Behaviors are analyzed from the difference between the left hand side and the right hand side of each rule for every object. Every object can have one or more behavior. There are 4 kinds of behaviors:
  - **Move:** If an object on the left hand side have different movement than the right hand side, this object has a Move behavior. For example, In the following rule Crate moves when Player approaches it.

    [> Player | Crate] -> [> Player | > Crate]
  - **Teleport:** An object is considered to have a Teleport behavior if its location in the rule changes from the left hand side to the right hand side. For example, In the following rule Crate changes position with Player on collision.

    [> Player | Crate] -> [Crate | Player]

– **Create:** If the number of a certain object on the left hand side is less than its number on the right hand side, then this object has a Create behavior. For example, In the following rule, Crate is created when Player moves to an empty place.

[> Player |] -> [Crate | Player]

– **Destroy:** If the number of a certain object on the left hand side is greater than its number on the left hand side, then this object has a Destroy behavior. For example, In the following rule, the three Crates are destroyed when they are aligned beside each other.

[Crate | Crate | Crate] -> [| |]

- **Minimum Number:** It is the maximum number of times for an object to appear in the left hand side of game rules. For example consider the following group of rules:

  [> Player | Crate] -> [> Player | > Crate]

  [> Crate | Crate] -> [> Crate | > Crate]

  The Crate object appeared in the both rules. The first rule the Crate object appeared once, while the second rule it appeared twice. This means the minimum number of Crates is two. This is not the case when an object have a Create behavior. Create rules are responsible for generating objects. The Minimum Number of the Create objects is updated to reflect the least number of appearances in the Create rules. For example the following rules have two Create rules (the first and the third).

  [> Player |] -> [Crate | Player]

  [> Crate | Crate] -> [> Crate | > Crate]

  [Gem | Crate | Gem] -> [Crate | Crate | Crate]

  The number of Crate objects in each rule are 0, 2, 1 respectively. In normal case, the minimum number of Crate object will be 2. Since Crate object have Create behavior (in both the first and the third rule) then the minimum number of objects will be zero instead.

- **Relations:** It is a list of all objects that appears in the same rule with a certain object. For example, In the following rules, Crate has relations with Player and Lava, Player has a relation with Crate, and Lava has a relation with Crate.

  [> Player | Crate] -> [> Player | > Crate]

  [> Crate | Lava] -> [|]

  Each object has a special list for the left hand side only beside the main Relations list.

4.1.2 Level Generator

The Level Generator is responsible for creating a level in the best possible way. Two approaches were used to generate levels. The following subsections will discuss each one of them.

4.1.2.1 Constructive Approach

Constructive Approach uses information from the Rule Analyzer to modify the Level Outlines. In this approach, several levels are generated using a certain algorithm and the best levels are selected. A pseudo code for the algorithm is presented in Algorithm 1.

Algorithm 1: Pseudo algorithm for the Constructive Approach

Data: level outline, rule analysis
Result: modified level outline

numberObjects = Get the number of objects for each object type;

levelOutline = Insert Solid Objects in the level outline;
levelOutline = Insert Winning Objects in the level outline;
levelOutline = Insert Player Object in the level outline;
levelOutline = Insert Critical Objects in the level outline;
levelOutline = Insert Rule Objects in the level outline;

return levelOutline;

The algorithm consists of two main part. The first part is responsible for determining the amount of objects that should be presented in the current level outline. The second part is responsible for inserting game objects in an intelligent way to the current level outline. The order of the insertion algorithm places the most important game objects first to ensure playability.

Algorithm 2 shows the process of calculating the amount of objects. The algorithm starts by determining the percentages for each object type. Each object type contributes by a percentage equal to its minimum number to make sure that all rules can happen. A cover percentage is calculated based on the number of critical and winning objects. The value of the cover percentage is inversely proportional with the summation of both critical and winning objects. Critical and Winning Objects are the main game objects, without them game may not be playable at all. The increase in their numbers causes the level to be more difficult and more complex. Having a small cover percentage when they are huge, makes sure the game is not very complex.

Algorithm 2: Get the number of objects algorithm

Data: level outline, rule analysis
Result: Number of Objects for each type

percentages[Winning Object] = Minimum Number[Winning Object 1] + Minimum Number[Winning Object 2];
if *Player Object is a Winning Object* **then**
    percentages[Winning Object] = 2;
end
percentages[Solid Object] = Number of different kinds of Solid Objects;
percentages[Critical Object] = Summation of the minimum number of all Critical Objects;
percentages[Rule Object] = Summation of the minimum number of all Rule Objects;
Divide all percentages by their total summation;

coverPercentage = 1 - percentages[Winning Object] - percentages[Critical Object];
totalNumber = coverPercentage * total free area in the level outline;
numberObjects = totalNumber * weights * percentages;
numberObjects[Player] = 1;

return numberObjects;

The following algorithms are responsible for inserting objects based on the numbers resulted from the previous part. Most of them always need to find the most suitable empty locations to insert the new Object. The most suitable location is calculated based on the features of the inserted object. If the object has a Move behavior, it should be inserted at spots with the most free locations around it. Otherwise any random free location is okay.

Algorithm 3 shows the insertion algorithm for Solid Objects. The algorithm inserts a random solid objects at a random empty space (as Solid Object has not a Move behavior) in the level outline. The algorithm is repeated for several times based on its number. The same idea is used for inserting Player Object in Algorithm 5 but for only one time.

Algorithm 3: Solid Objects Insertion Algorithm

Data: level outline, rule analysis, number of objects
Result: modified level outlines

while *numberObjects[Solid Object] > 0* **do**
    object = Get a random solid object;
    location = Get a suitable empty location;
    levelOutline[location] = object;
    numberObjects[Solid Object] -= 1;
end

return levelOutline;

Before inserting the Player to the level, Winning Objects should be inserted. Algorithm 4 is responsible for inserting Winning Objects into the level outline. The algorithm generates an equal amount of both winning objects except if any of these objects have a Create behavior. The amount of the generated Winning Objects must be a multiple of their minimum number to ensure that all rules can be applied. The first winning object is inserted at any suitable empty location, while the other must be inserted at the farthest suitable empty location. Inserting at a very far location ensures a more difficult level. If the Winning Rule is No then the second object will be inserted at the same location of the first object.

Algorithm 4: Winning Objects Insertion Algorithm

```
Data: level outline, rule analysis, number of objects
Result: modified level outlines
if Winning Objects have Create beahvior then
    minObject1 = Minimum Number(Winning Object 1);
    minObject2 = Minimum Number(Winning Object 2);
else
    minObject1 = Max(Minimum Number(Winning Object 1), Minimum
    Number(Winning Object 2));
    minObject2 = minObject1;
end
while numberObjects[Winning Object] > 0 do
    for 1 to minObject1 do
        location = Get a suitable empty location for Winning Object 1;
        levelOutline[location] = Winning Object 1;
        numberObjects[Winning Object] -= 1;
    end
    for 1 to minObject2 do
        if Winning Rule != No then
            location = Get the farthest suitable empty location for Winning Object 2;
        end
        levelOutline[location] = Winning Object 2;
        numberObjects[Winning Object] -= 1;
    end
end
return levelOutline;
```

Algorithm 5: Player Object Insertion Algorithm

```
Data: level outline, rule analysis, number of objects
Result: modified level outlines
location = Get a suitable empty location for the Player Object;
levelOutline[loction] = Player Object;
return levelOutline;
```

Algorithm 6 is responsible for inserting Critical Objects. Critical Objects are one of the most important objects in the game. As they are connected with both Player and Winning Objects. In some games, levels are not solvable without Critical Objects. For example, the following rules are from game called DestroyGame. The game goal is to destroy all Gem objects. Gem objects can be destroyed if they are aligned with 2 Crate objects. Crate objects can be pushed by the Player object.

[> Player | Crate] -> [> Player | > Crate]

[Crate | Gem | Crate] -> [| |]

From these rules Crate is a critical object as it is connected with Gem and Player objects. If there is no Crates in the level, the game will be unplayable. For that reason the algorithm ensures inserting the minimum Number of all the critical objects. The algorithm adds more critical objects according to its number. Each critical object has a chance to appear directly proportional with its Priority feature.

Algorithm 6: Critical Objects Insertion Algorithm

```
Data: level outline, rule analysis, number of objects
Result: modified level outlines
foreach object in Critical Objects do
    for 1 to Minimum Number of object do
        location = Get a suitable empty location;
        levelOutline[location] = object;
        numberObjects[Critical Object] -= 1;
    end
end
while numberObjects[Critical Object] > 0] do
    object = Choose a random Critical Object based on its priority;
    for 1 to Minimum Number of object do
        location = Get a suitable empty location;
        levelOutline[location] = object;
        numberObjects[Critical Object] -= 1;
    end
end
return levelOutline;
```

Same is done with Rule Objects in Algorithm 7. Random Rule Objects are inserted to the map based on their Priority feature.

Algorithm 7: Rule Objects Insertion Algorithm

```
Data: level outline, rule analysis, number of objects
Result: modified level outlines
while numberObjects[Rule Object > 0] do
    object = Choose a random Rule Object based on its priority;
    for 1 to Minimum Number of object do
        location = Get a suitable empty location;
        levelOutline[location] = object;
        numberObjects[Critical Object] -= 1;
    end
end
return levelOutline;
```

4.1.2.2 Genetic Approach

This method uses GA to evolve level outlines to playable levels. Elitism is used to ensure that the best levels are carried to the next generation.

Chromosome Representation: In this technique levels are represented as 2D matrix. Each location value represents all the objects at that location.

Genetic Operators: Crossover and Mutation are used to ensure better levels in the following generations. One point crossover is used where a point (x, y) is selected from the first chromosome and all previous rows (having smaller x) are swapped with the second chromosome. Mutation changes any random selected position using the following:

- **Creating an object:** a random object is selected to replace an empty position in the level. In "No" winning condition, the two winning objects are created at the same location.
- **Deleting an object:** a random object from the level is deleted.
- **Changing object position:** a random empty position is swapped with a non-empty one.

The mutation operation happens by subjecting the level outline to these 3 methods using different probabilities. The Creating and deleting an object have lower probability than the changing object position.

Initial Population: Three different techniques are used to generate an initial population for the GA. These techniques are:

- **Random Initialization:** The population is initialized as mutated versions of the empty level outline. This technique takes very long time to find a good designed level as it search in the whole space.
- **Constructive Initialization:** The population is initialized using the Constructive Approach algorithm. Using the algorithm tighten the search space so less time is required to find good playable levels.

- **Mixed Approach:** The population is initialized as a mixture between the Random Initialization and the Constructive Initialization. A portion is initialized using the Constructive Approach Algorithm and some mutated versions, while the other portion is the same like the Random Initialization. Using that algorithm ensure more diversity than the previous two approaches.

More details about the results of each algorithm will be discussed in Chapter 5.

4.1.3 Level Evaluator

Level Evaluator is responsible for evaluating the generated levels. Level evaluation is based on some global knowledge about the Puzzle Script Language and Puzzle Games. The easy way to evaluate a puzzle level is to measure its level playability. Level playability is achieved by using an automated player which will be discussed later. Playability is not enough to judge puzzle levels. Some other metrics must be used to ensure that the solution have more moves with some thinking ahead. Figure 4.2 shows several levels designed for Sokoban game where all are playable but some of them are more interesting than the others. The first level is a very easy level which can be solved in one move. The second level need more moves which is more interesting, but it has a straight forward solution. The last level is not an easy to solve, it needs some prior thinking which is more interesting than the previous two.

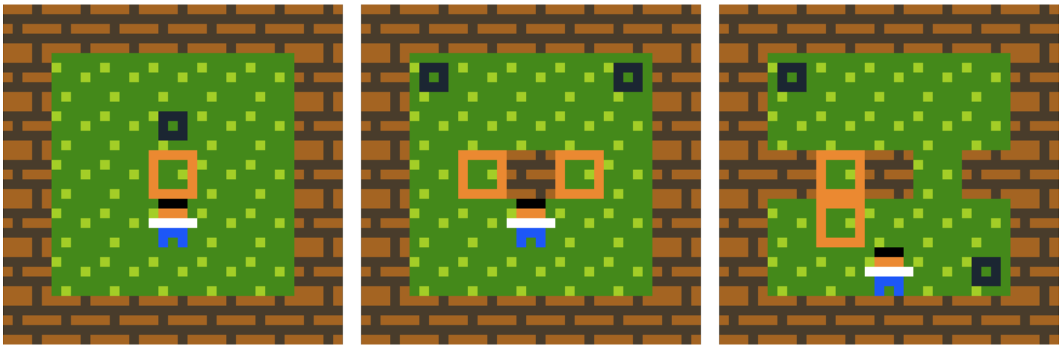

Figure 4.2: Examples on different levels for Sokoban game

The example shows the need for using heuristic measures to ensure more interesting levels.

4.1.3.1 Automated Player

Our Level Evaluator uses a modified version of the BestFS Algorithm as the automated player. BestFS Algorithm was introduced in Lim et al.[29] work. BestFS is similar to BFS algorithm but instead of exploring states sequentially, it sorts them according to a score generated from a fitness function. This causes the algorithm to explore the more important nodes first, helping it to reach the solution faster. As explained in Chapter 3, Lim et al. algorithm uses two metrics (as a fitness function) to evaluate each game state:

- **Distance between winning objects:** BestFS tries to either increase or decrease the distance between the winning objects according to the winning condition. The

"No" rule is the only rule that need to increase the distance, while the others need to decrease it. Figure 4.3 shows an example from Sokoban, where the distance between crates and targets is highlighted.

- **Distance between player and winning objects:** BestFS always tries to minimize the distance between the player and the winning objects. To achieve the first metric, the player should come near the winning objects. Figure 4.4 shows the same level from Sokoban, where the distance between player and winning objects (crates and targets) is highlighted.

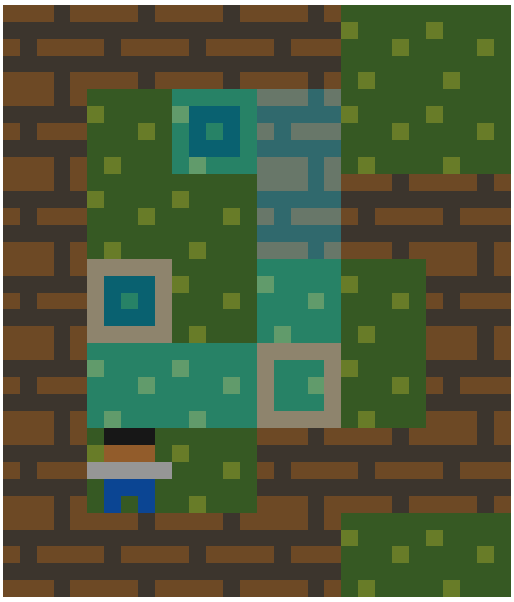

Figure 4.3: Example of distance between winning objects metric

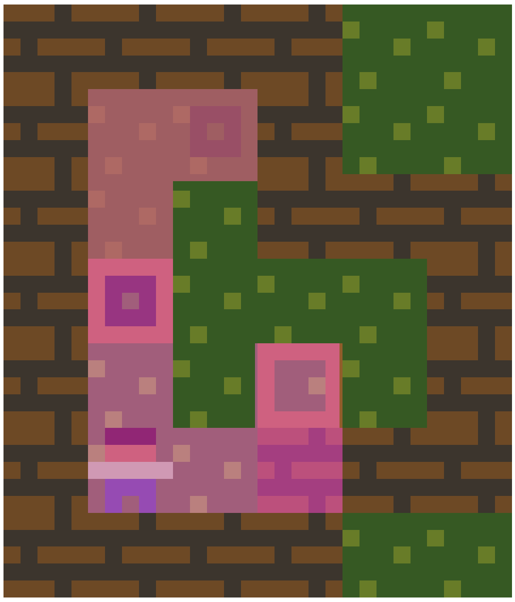

Figure 4.4: Example of distance between player and winning objects metric

These metrics works fine for all games where player is not one of the winning objects. When the player is one of the winning objects, the two metrics behaves in the same way. The player always try to move towards the winning objects regardless of any other game objects. For example, Figure 4.5 shows a level from a game called LavaGame. LavaGame is a puzzle game where the goal is to make the player reaches the exit. The path towards the exit is usually stuck by lava which can be destroyed by pushing a crate over it. According to the Lim et al. metrics, the player will try to move nearer to the exit by going left. This movement will not help the player to reach the exit, so the player will start wandering aimlessly trying to stumble across a movement sequence that can solve the level.

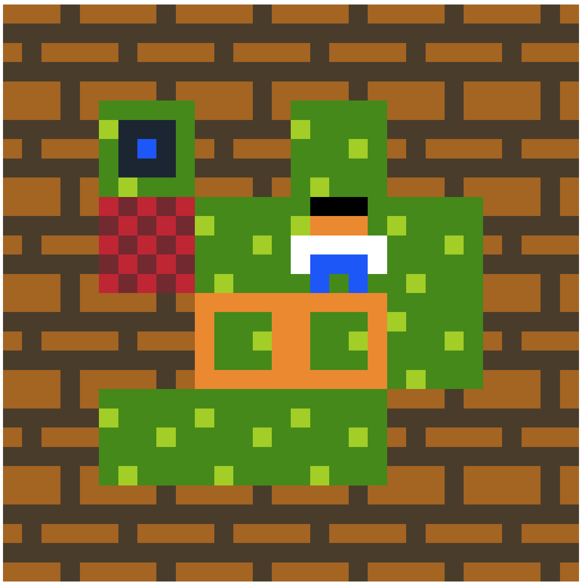

Figure 4.5: Example level from LavaGame showing the problem in the old metrics

Player's aim is to move crates towards the lava to unblock his path towards the exit. This aim is somehow explained in the game rules, so by further analysis the game rules we can know which objects need to be closer. Returning to our example about LavaGame, the game rules are stated in the following order:

[> Player | Crate] -> [> Player | > Crate]

[> Crate | Lava] -> [|]

The first rule says if there is a player and crate beside each other and the player moves toward it, the crate will also move. The second rule says if there is a crate and lava beside each other and the crate moves toward it, both crate and lava will be destroyed. In any proper game, rules must be applied before achieving the winning condition. Based on that fact, the distance between objects on the left hand side of the rules must be decreased. The relation between objects in the left hand side of the rules is captured by the Rule Analyzer Relations list for each game object. This distance is used as the new heuristic measure beside the original ones. The three metrics are weighted with respect to each other and used as a new fitness function to evaluate game states. The weights are chosen by experimentation to ensure the best results.

The output of the automated player is essential in evaluating the level quality. Four different values are returned which capture the way the automated player plays the level. These four values are:

- **The score for the best reached state so far:** The score is calculated using the first metric (Distance between winning objects). The score value is in range between [0, 1], where 0 means there is no winning objects, while 1 means the player reached the solution.
- **The movement sequence to reach the best state:** The automated player saves up all the movement happens to reach each state and returns the sequence that leads to the best state.

- **The number of states explored while searching for the solutions:** The automated player saves the number of states that it explores before terminating.
- **The number of rules that the game engine applies to reach the best state:** With each movement, some rules may apply through the game engine. This value returns the number of rules applied by the game engine to reach the best state.

The modified BestFS finds the solution faster than original one. More details will be explained in Chapter 5.

4.1.3.2 Heuristic Measures

Heuristic measures are calculated using a weighted function of six attributes. The function is described as the following:

$$F_{score} = 0.3 * P_{score} + 0.2 * L_{score} + 0.15 * N_{score} + 0.12 * B_{score} + 0.12 * R_{score} + 0.11 * E_{score}$$

where P_{score} is the Playing Score, L_{score} is the Solution Length Score, N_{score} is the Object Number Score, B_{score} is the Box Line Score, R_{score} is the Applied Rule Score, and E_{score} is the Exploration Score. The weights for each attribute are measured experimentally to reflect the importance of each features with respect to the others. The following points will further explain each of these scores.

- **Playing Score (P_{score}):** Playing score is used to ensure level playability. Instead of using a boolean value for playable or not. A float value is assigned for how much the player is near the solution. The first output of the automated player is used for that purpose. Making the domain more continuous (using float) helps in measuring the percentage of the level playability instead of using it as a constraint.

  Based on the work by Nielsen et al.[32] which proved that Do Nothing player is an important measure for good designed games. A score is calculated for the initial level state using the same way. This score is subtracted from the previous score. The Playing Score can be expressed by the following equation:

  $$P_{score} = S_{play} - S_{nothing}$$

  where S_{play} is the automated player score and $S_{nothing}$ is the Do Nothing player score.

- **Solution Length Score (L_{score}):** Figure 4.2 shows that interesting levels usually have more steps than the trivial ones. The first idea is to compare the length of the best movement sequence with a target value. This idea will not work as expected because solution length depends on the size of the level. For example, Figure 4.6 shows different levels from Sokoban and their corresponding solution length. Its obvious that the seconds level have longer solution because it has bigger area.

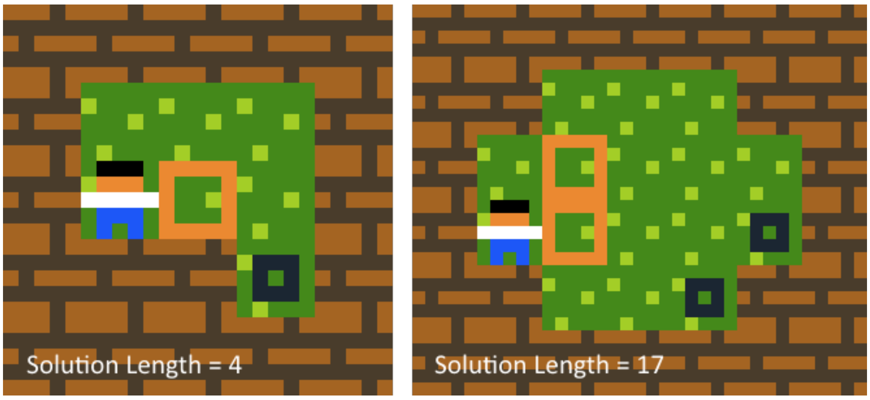

Figure 4.6: Examples of two sokoban levels with different area and solution lengths

From the previous example we can conclude that the solution length depends on the level area. Instead of using the solution length as the metric we used the ratio between the solution length and the level area. A mapping function is used to convert that number to a value in the range of [0, 1]. We analyzed 40 hand crafted levels with different area from 5 different games. A histogram is plotted for the ratio between the solution length and the level area and shown in Figure 4.7.

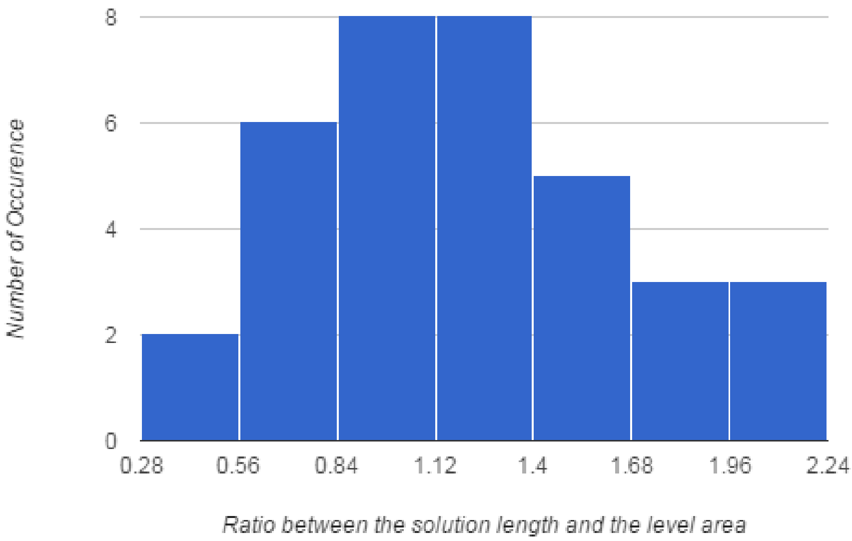

Figure 4.7: Histogram for the ratio between the solution length and the level area

The histogram seems to follow a Normal Distribution with $\mu = 1.221$ and $\sigma = 0.461$. Based on that, the Solution Length Score is expressed by the following equation:

$$L_{score} = Normal(\frac{L}{A}, 1.221, 0.461)$$

where $Normal(ratio, \mu, \sigma)$ is a normal distribution function, L is the solution length, and A is the level area.

- **Object Number Score (N_{score}):** The Object Number Score is calculated by the following equation:

$$N_{score} = 0.4 * N_{rule} + 0.3 * N_{player} + 0.3 * N_{winning}$$

  where N_{rule} is the Number of Rule Objects ratio, N_{player} is the Number of Players value, and $N_{winning}$ is the Number of Winning Objects value.

  - **Number of Rule Objects (N_{rule}):** In a good designed level, most of the rule objects should appear in the level to ensure there is a possibility of applying every rule. The number of times the object should appear in the level must be greater than or equal his minimum number property from the Rule Analyzer. A ratio is calculated between the number of objects greater than their minimum number property and the total number of the rule objects.

$$N_{rule} = \frac{N_{min}}{N_{max}}$$

    where N_{min} is the number of objects greater than their minimum number property and N_{max} is the total number of the rule objects.
  - **Number of Players (N_{player}):** The game should have only one player. If any level have a different value, the score will be zero.

$$N_{player} = \begin{cases} 1 & \text{one player object exists} \\ 0 & \text{otherwise} \end{cases}$$

  - **Number of Winning Objects ($N_{winning}$):** The number of the winning objects should be equal, unless one of the winning objects have "Create" behavior. Based on the previous condition, the score is set to either one or zero.

$$N_{winning} = \begin{cases} 1 & N_{winnObj1} == N_{winObj2} \text{ and no Create behavior} \\ 1 & \text{Create behavior exists} \\ 0 & \text{otherwise} \end{cases}$$

    where $N_{winObj1}$ is the number of the first winning object and $N_{winObj2}$ is the number of the second winning object.

- **Box Line Score (B_{score}):** It is similar to Taylor and Parberry[57] metric to find the farthest state. This metric calculates the number of unrepeated moves found in the solution and divide it by the solution length. The following equation represents it:

$$B_{score} = \frac{L_{unique}}{L}$$

  where L_{unique} is the number of unrepeated moves in the solution and L is the solution length.

- **Applied Rule Score (R_{score}):** Good level design involves applying game rules for a number of times to solve any level. The ratio between the number of applied rules to the solution length should be used for indicting good level design. Exaggerating in applying the rules results in boring level. Same can be said for the very low amount of applying. Figure 4.8 shows two levels from Sokoban with different solutions. The left level needs to apply Sokoban's rule for one time to solve the level, while the right needs to apply Sokoban's rule with every single step.

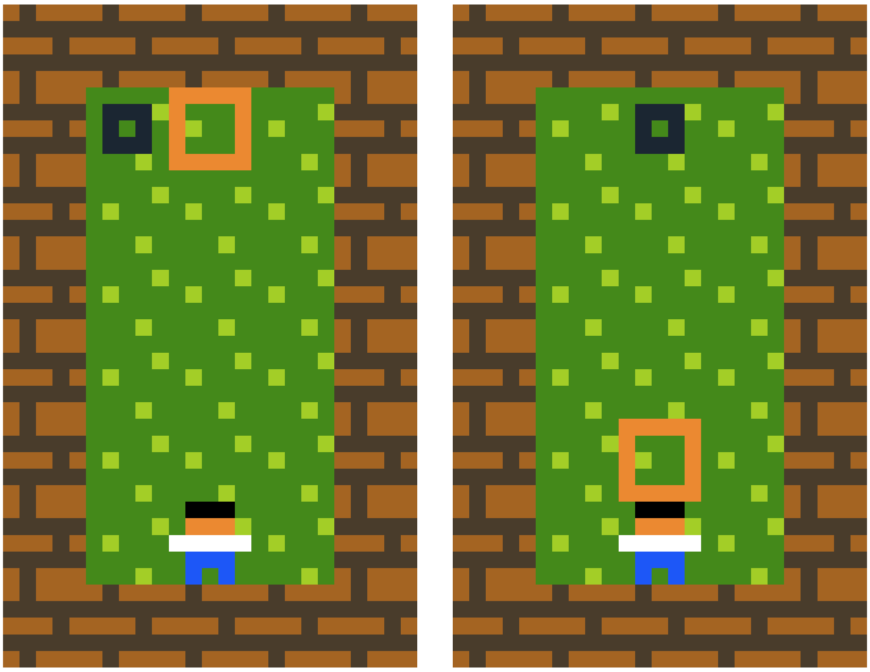

Figure 4.8: Example for two boring levels from Sokoban

To find the best ratio between both, We analyzed 40 hand crafted levels from 5 different games. A histogram is plotted for the ratio between the number of applied rules and the solution length and shown in Figure 4.9.

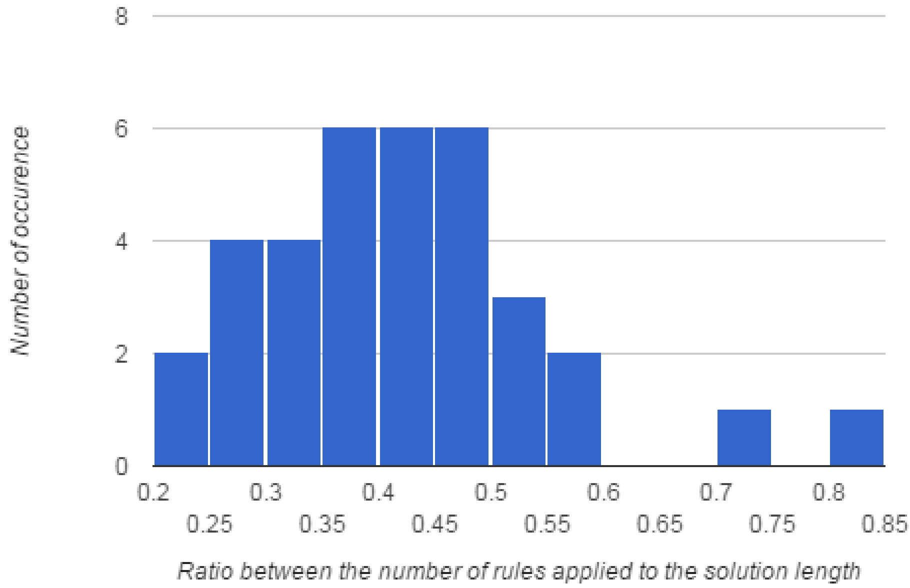

Figure 4.9: Histogram for the number of rules applied to the solution length

The histogram seems to follow a Normal Distribution with $\mu = 0.417$ and $\sigma = 0.128$. Based on that the Applied Rule Score can be expressed by the following equation:

$$R_{score} = Normal(\frac{R_{applied} \pm R_{none}}{L}, 0.417, 0.128)$$

where $Normal(ration, \mu, \sigma)$ is a normal distribution function, $R_{applied}$ is the number of applied rules, R_{none} is the number of applied rules with no action associated, and L is the solution length. The R_{none} is used to decrease the normal distribution value according to amount of rules applied at the beginning of the game with no action associated to decrease them from happening.

- **Exploration Score (E_{score}):** The increase in the number of explored states by the automated player means that the current level is not obvious to be solved directly by the automated player heuristics. This does not mean exploring huge space without finding a solution is better than exploring small number of states with a solution. The following equation express this idea:

$$E_{score} = \begin{cases} 0.75 + \dfrac{N_{explored}}{N_{max}} & \text{solution exists} \\ 0.5 & \text{no solution and } N_{explored} = N_{max} \\ 0 & \text{no solution and } N_{explored} < N_{max} \end{cases}$$

where $N_{explored}$ is the number of explored states and N_{max} is the maximum number of states the automated player can explore.

4.2 Rule Generation

As stated in the previous chapters, Rule Generation is the most difficult task in PCG. This work explores the idea behind generating new rules for the Puzzle Script engine. Unlike the work by Lim et al.[29], we are exploring Puzzle Script rules without fixing a certain level or a winning condition. Figure 4.10 shows a high level block diagram for the system. The system is a bit similar to the level generation system as the level generation system is a module inside the whole system.

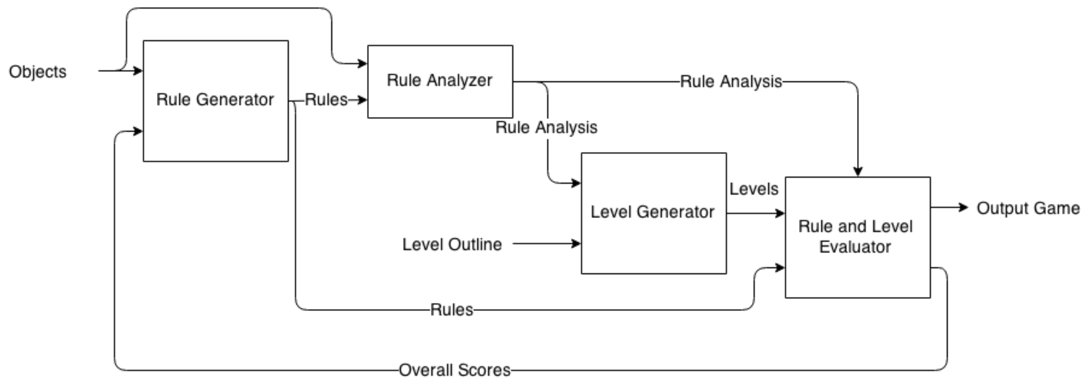

Figure 4.10: High level system block diagram for Rule Generation

The system starts by generating random rules for different games. These rules are analyzed using the same Rule Analyzer used in the Level Generation process. The output of the Rule Analyzer (Rule Analysis) with a level layout are fed to a Constructive Approach Level Generator. The generated rules and the resulted levels are subjected to an evaluator similar to the one used in the Level Generation process. The system evolves using the same GA operators from Lim et al. work[29]. In the following subsections we are going to discuss and explain every module in the block diagram.

4.2.1 Rule Generator

Rule Generator is responsible for generating and evolving rules. Rule generator uses GA for the evolution process. It is the same system used by Lim et al.[29] in his work.

Chromosome Representation: The chromosome is represented as a list of rules of maximum of four and a minimum of one and a single winning rule.

Genetic Operators: Crossover and Mutation are used to evolve the rules from one generation to the next. Crossover takes place only when the two chromosomes possess the same number of rules, the same number of tuples within each rule, and the same number of objects per tuple. There is two kind of crossover that can take place:

- **Hand Side Crossover:** Single point crossover is used to swap the whole right hand side between rules of different chromosomes.
- **Element Crossover:** Two point crossover is used to swap a single object from the first chromosome with the second one.

Mutation has no preconditions to take place except for a certain probability. There is five different mutation operators (mutators). Chromosomes are subjected to each one based on a certain probability. The five mutators are:

- **Rule size mutator:** changes the number of rules either by adding an empty rule or removing a rule.
- **Object mutator:** changes a random object in a rule with any random object.
- **Direction mutator:** changes the direction of a random object in a rule. It also can add or remove a direction for a random object.
- **Tuple size mutator:** change the number of objects inside the tuple by either deleting a random object or adding a random object with a random direction.
- **Hand size swap mutator:** swaps the left hand side with the right hand side of a rule.

These operators are not used over the winning condition. Winning condition have a separate crossover and mutation operators. Crossover is done using one point crossover to swap the winning objects and the winning rule. Mutation swaps any of the objects to a random object or change the winning rule to a random rule.

Initial Population: The population is initialized by mutated versions of an empty rule.

4.2.2 Rule Analyzer

Rule Analyzer is responsible for understanding the generated rules by the Rule Generator. The same Rule Analyzer from Section 4.1.1 is used here.

4.2.3 Level Generator

Level Generator is responsible for generating random levels for the generated rules. It uses the Constructive Approach algorithm described in Section 4.1.2.1.

4.2.4 Rule and Level Evaluator

Rule and Level Evaluator is responsible for giving a score for the generated games (rules and levels). It is a mixture between the Level Evaluator from Section 4.1.3 and Lim et al.[29] Evaluator. The game evaluation score can be expressed by the following equation:

$$RL_{score} = 0.3 * L_{score} + 0.7 * R_{score}$$

where L_{score} is the level score and R_{score} is the rule score. The rule score gets higher weight than the level score as it is the core of game generation, if the rules are bad, the game will not be playable.

4.2.4.1 Level Score

It uses the automated player described in Section 4.1.3.1 to give each generated level a score and the best scored level is the selected one. The level score is calculated using the same equation in Section 4.1.3.2 except for the Playability Score. The Do Nothing Score is removed from Playability Score calculations. The Do Nothing Score removal gives an equal chances to all the winning conditions. For example, the No rule always have a Do Nothing Score equal to zero, while other rules have a value. Leaving the Do Nothing score, the generator will always favor the No rule over other rules.

4.2.4.2 Rule Score

It is divided into 2 different parts rule heuristics score and rule validity score. The rule score can be expressed by the following equation:

$$R_{score} = 0.5 * H_{score} + 0.5 * V_{score}$$

where H_{score} is the rule heuristic score and V_{score} is the rule validity score.

Rule Heuristics Score (H_{score}): It is similar to Lim et al.[29] heuristics. It is a group of heuristics that ensures higher chance of playability for the generated games. The Rule Heuristic Score can be expressed by the following equation:

$$H_{score} = \frac{1}{8} * (P_{score} + C_{score} + U_{score} + W_{score} + WO_{score} + PL_{score} + M_{score} + D_{score})$$

where P_{score} is the player exists score, C_{score} is the critical path exists score, U_{score} is the useless objects score, W_{score} is the winning condition validity score, WO_{score} is the winning object in the rules score, L_{score} is the player in LHS score, M_{score} is the player movement score, and D_{score} is the direction consistency score.

- **Player exists score (P_{score}):** It ensures that a player exists in any of the game rules. The player object must exist in the rules to have some game interaction. It can be expressed by the following equation:

$$P_{score} = \begin{cases} 1 & \text{player exists in the rules} \\ 0 & \text{otherwise} \end{cases}$$

- **Critical path exists score (C_{score}):** It ensures the existence of a path between the player object and at least one of the winning objects in the Relations list. Having a path ensures that the game can be finished by moving with the player. It can be expressed by the following equation:

$$C_{score} = \begin{cases} 1 & \text{citical path exists} \\ 0 & \text{otherwise} \end{cases}$$

- **Useless objects score (U_{score}):** The amount of useless objects found in the game rules. Useless objects are rule objects that are neither connected to a player or a winning condition in the Relations list. Useless objects are not affected by any player action so the score is inversely proportional to their number. The score can be expressed by the following equation:

$$U_{score} = \frac{1}{N_{useless} + 1}$$

  where $N_{useless}$ is the number of useless objects.

- **Winning condition validity score(W_{score}):** The winning objects should have some features to ensure game playability. These features differ according to the winning condition. No rule requires at least one of the winning object having a Delete, Move, or Teleport behavior, while other winning rules (All and Some) requires Create, Move, or Teleport behavior. This score can be expressed by the following equation:

$$W_{score} = \begin{cases} 1 & \text{winning object have one of the required features} \\ 0 & \text{otherwise} \end{cases}$$

- **Winning object in the rules score (WO_{score}):** At least one of the two winning objects should appear in any rule. This score can be expressed using the following equation:

$$WO_{score} = \begin{cases} 1 & \text{at least one winning object exists in the rules} \\ 0 & \text{otherwise} \end{cases}$$

- **Player in LHS score (PL_{score}):** Player should exists on the left hand side of at least one rule. This score can be expressed by the following equation:

$$PL_{score} = \begin{cases} 1 & \text{player exists in LHS of at least one rule} \\ 0 & \text{otherwise} \end{cases}$$

- **Player Movement (M_{score}):** Player should have at least one movement action associated with it in any rule to affect other game objects. This score can be expressed by the following equation:

$$M_{score} = \begin{cases} 1 & \text{player movement exists} \\ 0 & \text{otherwise} \end{cases}$$

- **Direction consistency (D_{score}):** Rules should have the same movement action across the whole rule to ensure human playability. The score is inversely proportional to the number of different movements in the rules. This score can be expressed by the following equation:

$$D_{score} = \frac{1}{D_{unique}}$$

  where D_{unique} is the average number of unique movement for all rules.

Rule Validity Score (V_{score}): It ensures that no errors are found in the generated rules. Errors can be detected from Puzzle Script engine during the compilation process. This score can be expressed using the following equation:

$$V_{score} = \begin{cases} 1 & \text{no errors} \\ 0 & \text{otherwise} \end{cases}$$

Chapter 5: Results and Evaluation

This chapter shows the results of the different techniques of level generation and rule generation. The resulted levels and games of the system are published on a website [1] to collect human feedback. It shows also a comparison between the system results with collected human statistics. In the following sections we start describing the input for level and rule generation followed by description of our experiments. By the end compare its results with collected human statistics.

5.1 Automated Player

Section 4.1.3.1 introduces new metric to solve the problem of having Player as one of the winning objects. Several experiments are done to detect the best possible weights for the three metrics and comparing the results with Lim et al.[29] previous metrics.

The new player give different weights to each one of the three metrics. The distance between winning objects metric is the most important metric, other metrics contributes with less score (50% decrease). The distance between player and winning objects metric contributes with a lower percentage than the new metric (50% lower). This decrease is due to the fact that one of the winning objects is a rule object. The new metric measures the distance between the rule objects. This means this object contributes in both metrics at same time.

The following section will describe the different games and levels used to test both metrics followed by different experiments to measure the difference between both metrics.

5.1.1 Input Description

Forty handcrafted levels from five different games were used to test our the introduction of the new metric. The five games where completely different to ensure covering most of behaviors. Levels are also designed with different sizes and different ideas to cover different design aspect.

The five games are Sokoban, LavaGame, BlockFaker, GemGame, and DestroyGame.

- **Sokoban:** The goal of the game is to place every single crate over a certain position. Player is supposed to push crates to achieve that goal.
- **LavaGame:** The goal of the game is to reach the exit. Most of the time the path towards the exit is blocked by lava. Player is supposed to move crates over lava to clear his way.
- **BlockFaker:** The goal of the game is to reach the exit. The path towards the exit is blocked by lots of crates. Player should push the crates to align them vertically or horizontally. Every three aligned crates are removed clearing the path towards the exit.

[1] http://www.amidos-games.com/puzzlescript-pcg/

- **GemGame:** The goal of the game is to place at least one gem over one of several locations. Player can create gems by pushing crates. Every three aligned crates are replaced with a single diamond instead of the middle crate.
- **DestroyGame:** The goal of the game is to clear every single gem. Gems can be destroyed when they are aligned with two other crates vertically or horizontally. Player should push crates to reach that goal.

Each of the previous games have different rules, goals, and objects.

5.1.2 Comparing Different Players

Lim et al.[29] automated player plays all the forty levels and it is compared with the introduced automated player in this work. Figure 5.1 shows the average number of states each player explored in each game before reaching goal.

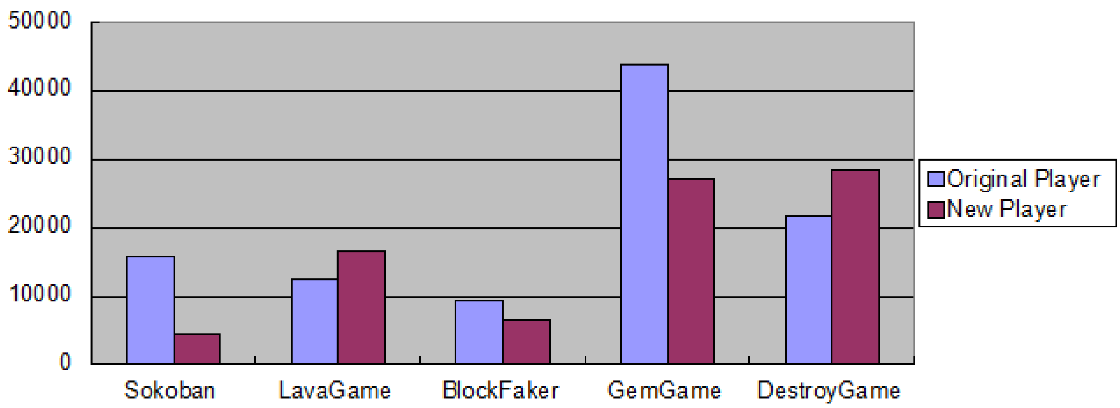

Figure 5.1: Comparison between the number of explored states for different automated players

The new player outperforms original player in Sokoban game and Gem game but its almost the similar in the rest of the games. In games where the player is one of the winning objects, the new player just performed slightly better in BlockFaker, while LavaGame is slightly worse. The same happens with DestroyGame. The main reason behind that is the presence of Destroy behavior as the core of the game. For example, Figure 5.2 have two crates and two lava, the metric measures the average distance between each crate and all lava. If the lower lava is destroyed first that remain us with one box with one lava which makes the metric a little bit worse than before (only long distance remains). The system will explore more states that will not lead to destroy of the lower lava.

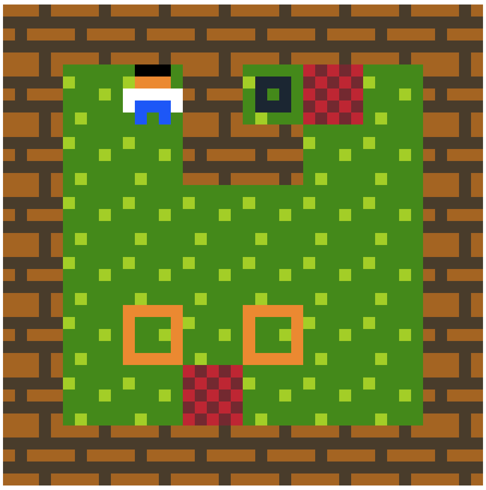

Figure 5.2: LavaGame level cause the new player to do worse

Another drawback of the new metric is when levels have lots of unused objects. The new metric will make the player explores these objects which increase the number of explored states before reaching the goal object. This drawback is not so important as the main goal is to generate levels not solve them. The presence of unused objects may cause increase of the difficulty of generated levels.

The average solution length of each game is presented in Figure 5.3. The new player produces slightly an overall shorter solutions than the original player. Comparing both Figure 5.1 and Figure 5.3 there is a correlation between the average number of explored states and the average solution length except for Sokoban. Sokoban doesn't follow same pattern due to being abstract with very small amount of objects and just one rule. This abstraction is the main reason both players can reach the goal in almost the same amount of steps.

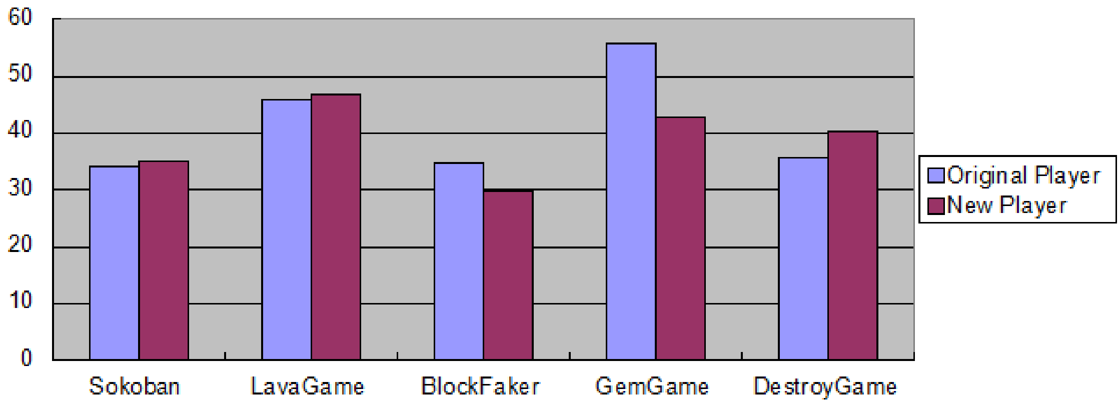

Figure 5.3: Comparison between the average solution length for different automated players

Table 5.1 shows the correlation value between the average solution length and the average explored states by the two different players and the correlation is calculated at the bottom of the table. The values in the table is the difference between the original player and the new player. Positive values indicates the new player is better than the original and vice versa.

The average solution length	The average explored states
-0.875	11127.625
-1	-3914
5	2736.625
12.875	16763.625
-4.75	-6696.75
Correlation	0.77127

Table 5.1: Correlation between the average explored states and the average solution length

5.2 Level Generation

This section shows the results of different techniques introduced in Chapter 4 for generating different levels. Automated player is used with fixed number of iteration. Automated player takes very long time to play the level. The automated player is limited to 5000 explored states to ensure fast execution and level quality.

The following section will discuss the required input data for the system to work, followed by the results for each of different approach used.

5.2.1 Input Description

From Chapter 4, level generation needs a game description and some level layouts. Five different games were tested with eight different level layouts.

5.2.1.1 Game Descriptions

The five different games are the same games described in Section 5.1.1. Sticking with same games as they covers different behaviors and winning rules.

- **Sokoban:** The game have four main objects (Player, Crate, Target, and Wall). Crate and Target are the winning objects with Crate having a Move behavior. Wall is a solid object. The game has an All winning rule.

- **LavaGame:** The game has five main objects (Player, Lava, Crate, Target, and Wall). Player and Target are the winning objects. Lava and Crate are normal rule objects with Destroy behavior. Crate has a Move behavior too. Wall is solid object. The game has an All winning rule.

- **BlockFaker:** The game has five main objects (Player, Crate, Stopper, Target, and Wall). Player and Target are the winning objects. Stopper and Crate are normal rule objects with Crate having Move and Destroy behaviors. Wall is solid object. The game has an All winning rule.

- **GemGame:** The game has five main objects (Player, Crate, Gem, Target, and Wall). Gem and Target are the winning objects with Gem having Create behavior. Crate is a critical rule object with Move and Destroy behaviors. Wall is a solid Object. The game has a Some winning rule.

- **DestroyGame:** The game has five main objects (Player, Crate, Gem, Target, and Wall). Gem and Target are the winning objects with Gem having Destroy behavior. Crate is a critical rule object with Move and Destroy behaviors. Wall is a solid object. The game has a No winning rule.

5.2.1.2 Level Layouts

The second input for the system is the level layouts. Eight different level layouts are used to generate levels. Figure 5.4 shows these layouts. The layouts have different sizes and different internal structure to test the generator with different inputs. The biggest level layout is $8x8$ tiles because as the level size increase, the system takes longer time to generate a good level. The generation time is restricted so less interesting levels are generated. Human statistics proves this point in certain games.

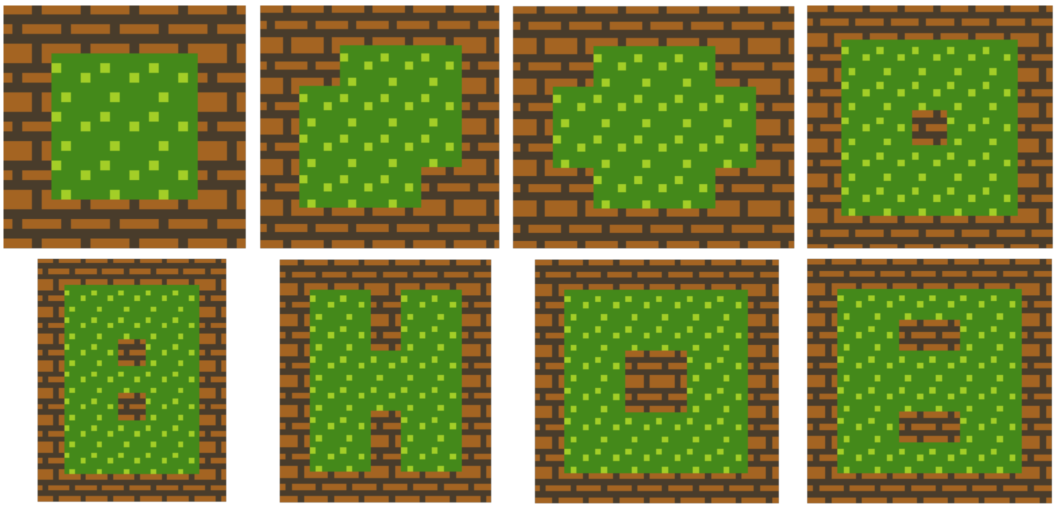

Figure 5.4: Different level layouts for level generation

5.2.2 Constructive Approach Results

Hundred level is generated using Constructive Algorithm. These levels are evaluated and the best two levels are selected. This approach is repeated for the five different games and applied over the eight different layouts. The total amount of generated levels are 80 levels. Figure 5.5 shows examples of some of generated levels for different games. All the generated levels can be played online[2].

[2]http://www.amidos-games.com

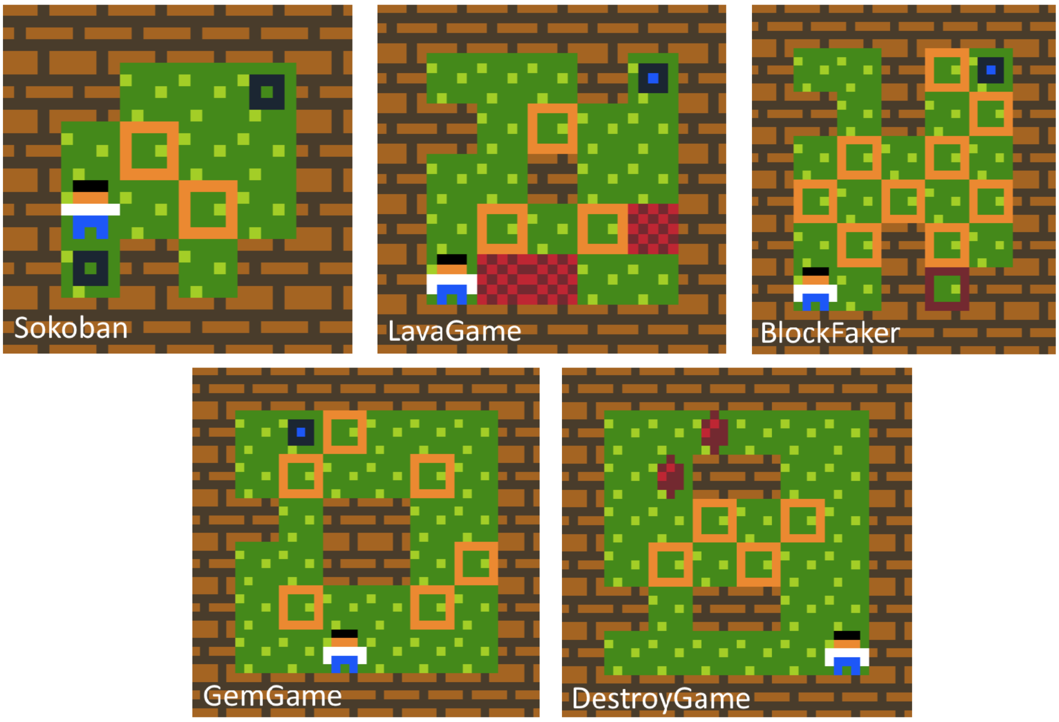

Figure 5.5: Examples of the generated levels using constructive approach

Out of the 80 levels only 15% are unplayable by the automated player but having a very high score. By testing these levels by human users only 10% of these are completely unplayable while the rest 5% are very challenging levels. Table 5.2 shows the score for all the generated levels using the level evaluator and human evaluation.

There is a huge correlation between the generated scores and the human scores for Sokoban and BlockFaker games than any other games. The reason behind that is the automated player plays these games better than others using 5000 states (refer to Figure 5.1). LavaGame has the most unplayable levels, because these unplayable games have high values in other heuristics compared with its playability score. GemGame and DestroyGame have a very small correlation because they are very hard games so the automated player always prefer the easy levels than hard ones. The reason behind that is the playability metric has huge effect on the final score as Do Nothing Score is always zero which is not the case with LavaGame and BlockFaker.

Sokoban		LavaGame		BlockFaker		GemGame		DestroyGame	
Automated Player	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**
0.616362	0.6	0.62840621	0	0.67445767	0.533333333	0.770599632	0.8	0.8232172	0.8
0.616362	0.6	0.5876679	0.533333333	0.6232392	0.533333333	0.770599	0.8	0.8232172	0.8
0.69734	0.768421053	0.5896824	0	0.65679029	0.533333333	0.9577728	0.8	0.9098534	0.8
0.694114	0.705263158	0.5821967	0.6	0.650513102	0.533333333	0.95777289	0.8	0.9095344	0.8
0.685905	0.811111111	0.62574735	0.466666667	0.60716514	0.6	0.96990456	0.6	0.914115	0.8
0.671965	0.8125	0.61386285	0.6	0.59945915	0.733333333	0.95356176	0.8	0.8975684	0.8
0.68588	0.677777778	0.6865308	0.6	0.743085297	0.666666667	0.85842389	0.6	0.9479144	0.6
0.67818	0.711111111	0.662806064	0.6	0.712869944	0.733333333	0.8531832	0.6	0.9418275	0.6
0.676906	0.694736842	0.7116651	0.533333333	0.750199824	0.866666667	0.91531076	0.6	0.9259153	0.6
0.6724738	0.705263158	0.6679423	0	0.74666923	0.8	0.8580357	0.6	0.9144401	0.8
0.70278	0.688888889	0.651134564	0.666666667	0.6585878	0	0.87784682	0.4	0.935399	0.6
0.698841	0.866666667	0.63963322	0	0.6572663	0.733333333	0.8774374	0.6	0.9293	0.6
0.659504	0.633333333	0.77533756	0.9	0.783839585	0.8	0.9196736	0.8	0.947915	0.6
0.6530206	0.705882353	0.67766235	0.6	0.7277456	0.8	0.8669255	0.8	0.9369172	0.8
0.681864	0.788888889	0.69552052	0	0.7283029	0.733333333	0.91133042	0.6	0.9449218	0.6
0.671696	0.766666667	0.667115002	0	0.7209062	0	0.89999055	0.8	0.93881904	0.8
Correlation	0.6634913	**Correlation**	0.2545798	**Correlation**	0.2734119	**Correlation**	-0.0026173	**Correlation**	-0.560329

Table 5.2: Automated Scores vs Human Scores for constructive approach

The constructive technique is a very fast technique that can be used for level generation in real time depending on the time of the automated player used. Although the algorithm ensures high level playability, it limits the search space for potential levels producing similar generated levels. The search space is limited due to the following:

- The number of objects are always limited by the size of the free areas in the map. That's why the two generated levels always have same number of objects.
- The moving object should be placed at the most free place. That's why the moving objects are always placed at certain places in a certain order.
- The second winning object is always at the farthest distance from the first object. That's why the second winning object is always aligned with level borders refer to Figure 5.5.

Smaller levels are subjected to higher similarity that bigger size levels. Out of the 80 levels only 8.75% are the same and all of them happens in the first three layouts specially the first level.

5.2.3 Genetic Approach Results

Genetic approach is used to give the system the ability to modify the generated levels to improve the playability and variety of games. Genetic algorithm is used for 20 generations with population of size 50. The crossover rate is around 70% and mutation rate is around 10%. The best two chromosomes of each level layout is published.

The maximum fitness of the GA is presented in Figure 5.6. Its clear that the best chromosomes disappear after few generations because they have really near fitness. The changes are not so huge but to ensure having best chromosome Elitism is used with probability equals 2%.

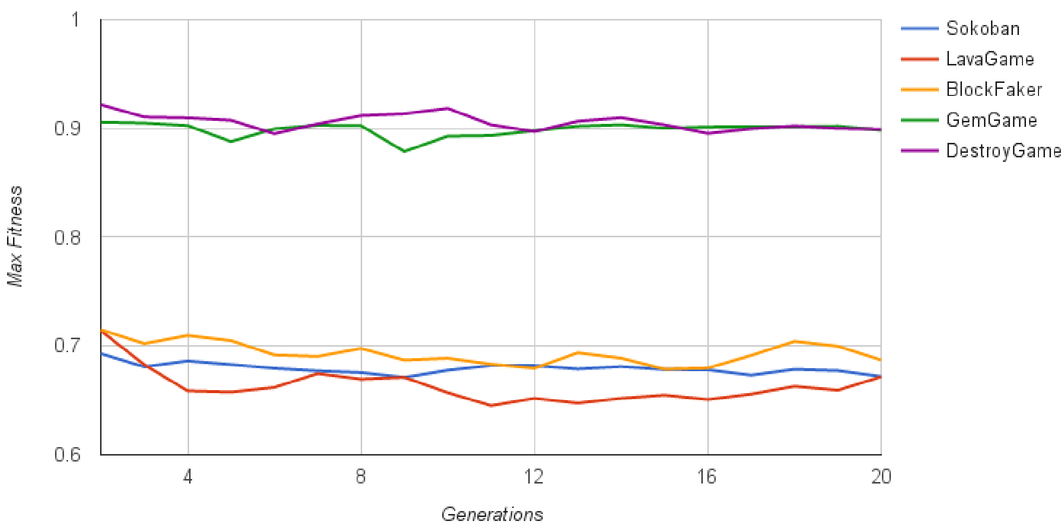

Figure 5.6: Maximum fitness for GA without Elitism

The drawback of using elitism that in most of the time the second best chromosome is the same as the best one which creates lots of redundant levels.

5.2.3.1 Random Initialization

Different initialization methods are tested alone and against each other. The first method is Random Initialization described in Section 4.1.2.2. Figure 5.7 and Figure 5.8 shows the evolution of maximum and average fitness respectively. Sokoban is the best evolved game in all of them. That's due to the small amount of objects needed to have a playable level, while GemGame requires pretty tough behavior to be playable that is why it take very long time to evolve. Although DestroyGame have a higher score than Sokoban but it has less quality. The DestroyGame has a zero Do Nothing score, while Sokoban always has a high Do Nothing score (refer to Table 5.2).

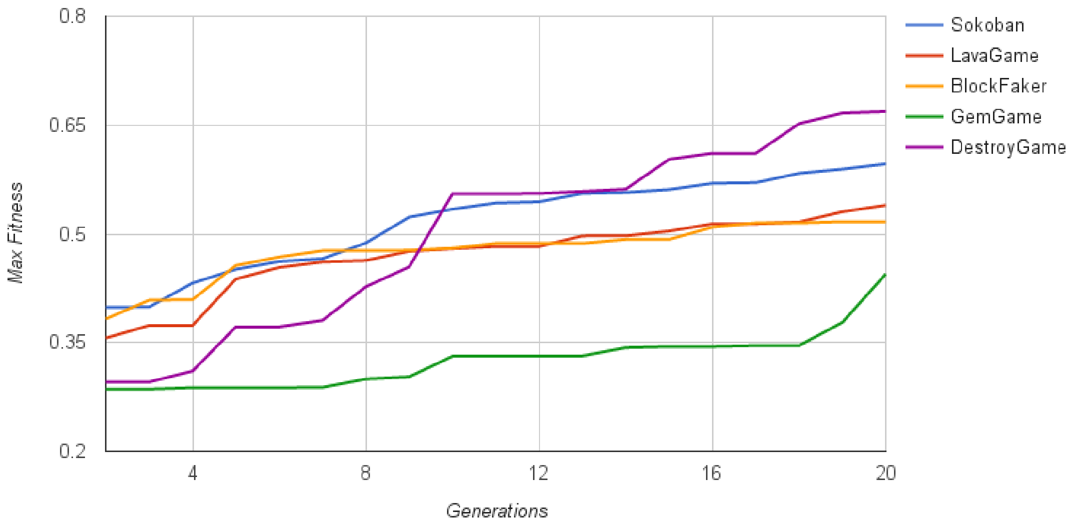

Figure 5.7: Maximum fitness for GA with random initialization

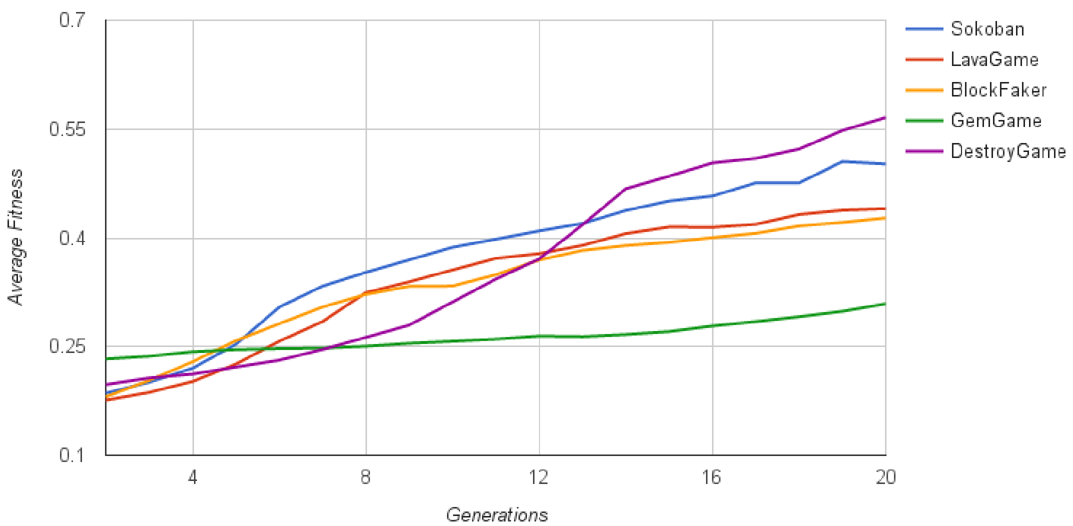

Figure 5.8: Average fitness for GA with random initialization

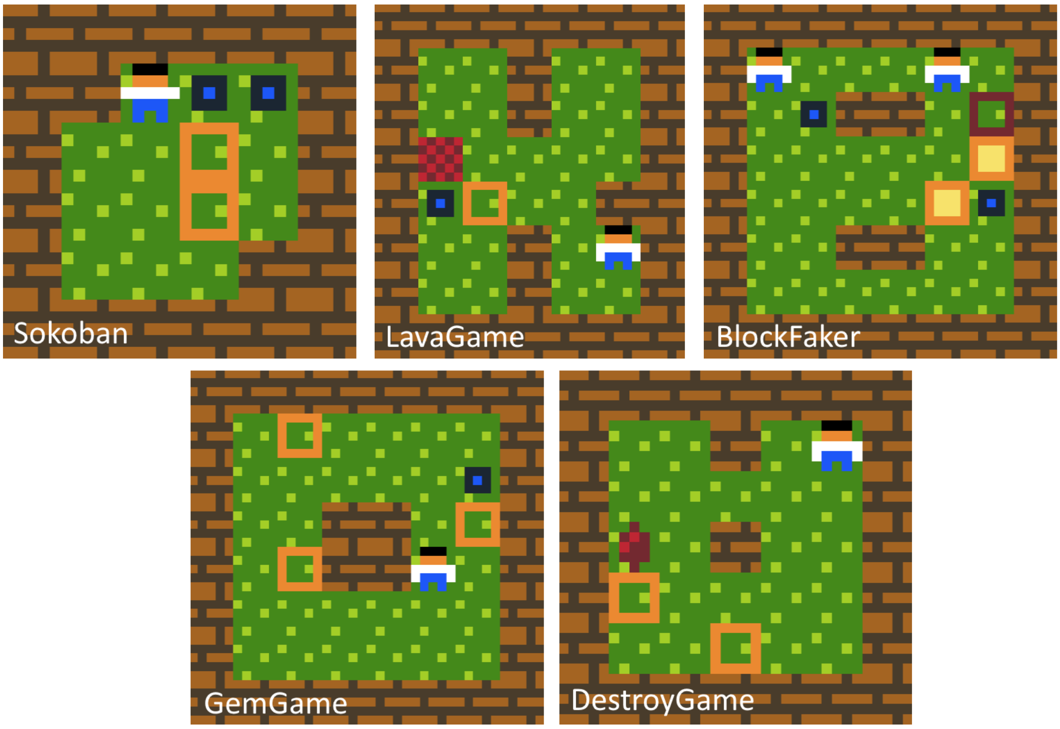

Figure 5.9: Examples of the generated levels using GA with random initialization

Figure 5.9 shows a group some of the different levels using GA with random initialization. These levels are subjected to human players to test the playability and quality of each generated level. About 42.5% of the generated levels are repeated levels and 75% of them are playable although the automated player reported 26.25% unplayable levels. Table 5.3 shows the correlation between the automated score and human scores. Most of the unplayable levels are found in the GemGame and DestroyGame, while most playable levels are LavaGame and BlockFaker. The reason behind that is that winning condition for BlockFaker and LavaGame is more easier than GemGame and DestroyGame as they required a certain number of objects to finish it.

The correlation is the highest at GemGame and DestroyGame because most of the generated levels are unplayable and their best state is the starting state. That is different from unplayable levels in constructive approach where it gets a higher score which indicates the percentage of level playability.

Sokoban		LavaGame		BlockFaker		GemGame		DestroyGame	
Automated Player	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**
0.472927827	0.566666667	0.55654906	0.9	0.56366794	0.4	0.2895113	0	0.823277724	0.7
0.472927827	0.566666667	0.55654906	0.9	0.56366794	0.4	0.2895113	0	0.823277724	0.7
0.66605524	0.630769231	0.4829044	0.4	0.58819125	0.4	0.2852678	0	0.285267846	0
0.66605524	0.630769231	0.4829044	0.4	0.58819125	0.4	0.2852678	0	0.285267846	0
0.620078	0.892307692	0.58316514	0.5	0.539436756	0.4	0.284260212	0	0.296260212	0
0.601255799	0.892307692	0.58316514	0.5	0.539436756	0.4	0.284260212	0	0.296260212	0
0.57708091	0.516666667	0.55400489	0.4	0.50191485	0.4	0.3350501	0	0.75341078	0.4
0.57708091	0.516666667	0.55400489	0.4	0.50191485	0.4	0.3350501	0	0.75341078	0.4
0.5720216	0	0.4216133	0.4	0.523848613	0.4	0.346181665	0	0.87578933	0.6
0.5720216	0	0.4216133	0.4	0.46675503	0.4	0.33422192	0	0.87108038	0.4
0.604114365	0.44	0.57729139	0.6	0.50550698	0.5	0.282968	0	0.865837091	0.6
0.604114365	0.44	0.57729139	0.6	0.50550698	0.5	0.282968	0	0.858546414	0.6
0.638117335	0.633333333	0.569474277	1	0.46041324	0.4	0.8438745	0	0.934646319	0.6
0.638117336	0.633333333	0.569474277	1	0.46041324	0.4	0.8438745	0.4	0.934646319	0.6
0.620584613	0.771428571	0.5843821	1	0.54673306	0.4	0.90855179	0.8	0.84207603	0.8
0.620584613	0.771428571	0.5843821	1	0.53355297	0.8	0.90855179	0.6	0.52655178	0
Correlation	0.281768023	**Correlation**	0.557156502	**Correlation**	-0.0011142	**Correlation**	0.824985501	**Correlation**	0.916474873

Table 5.3: Automated Scores vs Human Scores for GA with random initialization

5.2.3.2 Constructive Initialization

The problem with Random Initialization that it needs very huge amount of generation to converge to a playable challenging levels. Using constructive algorithm to initialize the GA ensures better playability as 90% of the generated level output is playable levels. GA will ensure the increase of that playability to reach 100% and it expands the search space finding more diverse results than Constructive Approach only. Figure 5.10 and Figure 5.11 show the evolution of max fitness and average fitness across the generations. The slow increase in the max fitness score is due to two reasons:

- The high quality of the generated levels by the constructive algorithm.
- The limited search space of the constructive algorithm.

The second point is the main reason why the average score drops at the beginning. At the beginning, GA expands the search space using the Mutation operators to find different levels. Crossover has a little bit effect as all the levels have same formate of the constructive algorithm. The same reasons above makes the generated levels very similar to the constructive algorithm output. For examples, Sokoban levels have crates set at the most empty spaces, ending targets on the borders, and ...etc. Even the unplayable levels seems like an alternated versions of the constructive algorithm levels.

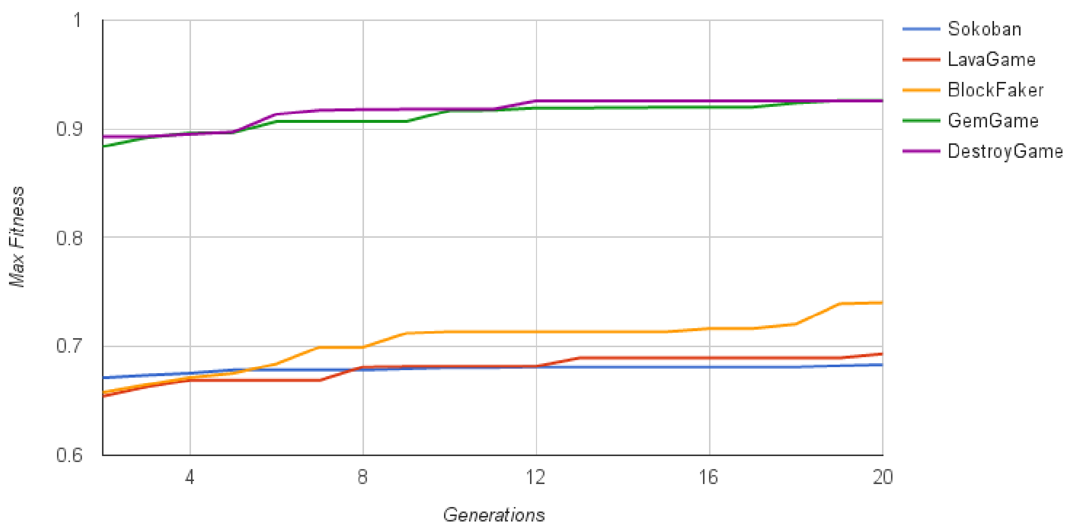

Figure 5.10: Maximum fitness for GA with constructive initialization

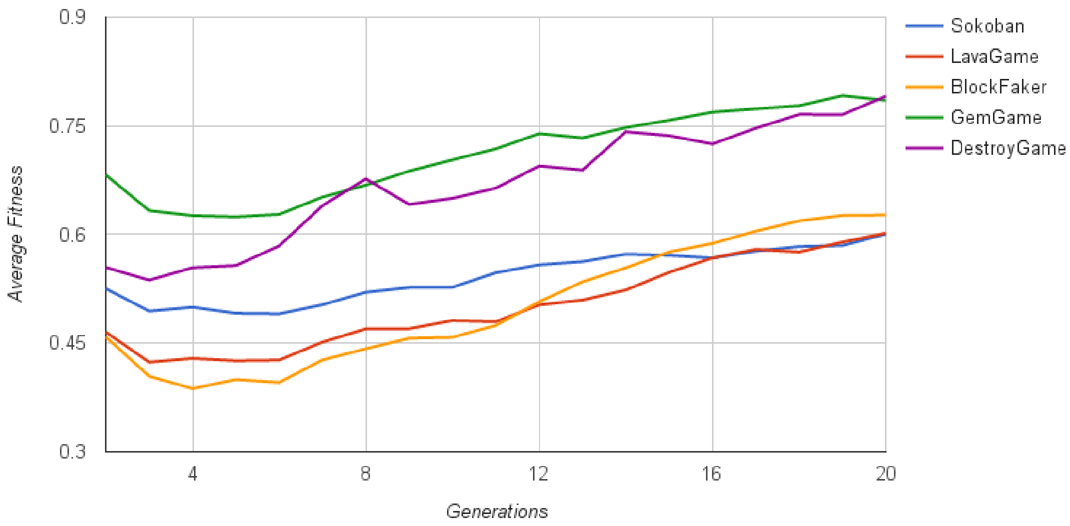

Figure 5.11: Average fitness for GA with constructive initialization

Figure 5.12 shows some generated levels. 46.25% of the generated levels are similar due elitism and small search space. All the levels generated are playable. Table 5.4 shows the automated player score with human scores and calculates the correlation between both of them. Same small correlation happens in the DestroyGame and GemGame due to its difficulty.

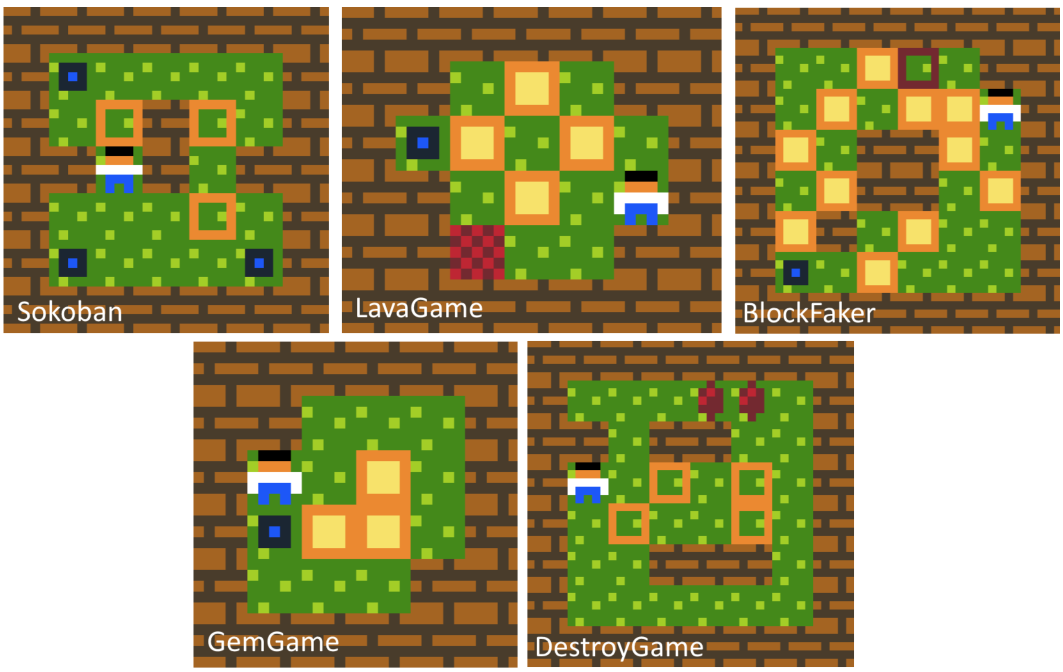

Figure 5.12: Examples of the generated levels using GA with constructive initialization

Sokoban		LavaGame		BlockFaker		GemGame		DestroyGame	
Automated Player	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**
0.616362	0.6	0.6191616	0.5	0.674457	0.6	0.8730908	0.8	0.8769583	0.8
0.615881	0.52	0.6191616	0.5	0.674457	0.6	0.8730908	0.8	0.8769583	0.8
0.71161371	0.766666667	0.6521912	0.6	0.640292	0.6	0.957772	0.8	0.927642	0.6
0.71161371	0.766666667	0.6121912	0.6	0.640292	0.6	0.957772	0.8	0.927642	0.6
0.6813152	0.6	0.6261598	0.8	0.613533	0.6	0.9541082	0.6	0.9201257	0.8
0.6813152	0.6	0.6260663	0.6	0.60379	0.6	0.9541082	0.6	0.9201257	0.8
0.692882	0.88	0.650007	0.7	0.75497	0.8	0.92124517	0.6	0.94662662	0.6
0.692882	0.88	0.650007	0.7	0.75497	0.8	0.92124517	0.6	0.94332112	0.6
0.690599	0.6	0.765587	0.9	0.80884	0.8	0.9003137	0.6	0.928826	0.8
0.690599	0.6	0.765587	0.9	0.80884	0.8	0.9003137	0.6	0.928826	0.8
0.702111	0.666666667	0.80019	0.7	0.827012	0.9	0.9340367	0.8	0.9418921	0.6
0.69881	0.666666667	0.80019	0.7	0.827012	0.9	0.9340367	0.8	0.9418921	0.6
0.672561	0.766666667	0.7755379	0.6	0.810844	0.8	0.9438273	0.6	0.93639401	0.8
0.672561	0.766666667	0.7755379	0.6	0.766883	0.8	0.9438273	0.6	0.93639401	0.8
0.703393	0.733333333	0.697292	0.6	0.817545	0.8	0.923987	0.6	0.9321509	0.8
0.703393	0.733333333	0.697292	0.6	0.817545	0.8	0.923987	0.6	0.9321509	0.8
Correlation	0.515628369	**Correlation**	0.42035216	**Correlation**	0.941391762	**Correlation**	-0.128466	**Correlation**	-0.465830

Table 5.4: Automated Scores vs Human Scores for GA with constructive initialization

5.2.3.3 Mixed Initialization

The generated games are 100% playable but most of them still have same structure of The constructive algorithm. To ensure more diversity, Mixed initialization is used. 25% of population is initialized using constructive algorithm, another 25% with mutated versions of levels from constructive algorithm, while the rest are mutated versions of level layouts. The mutated levels are the result of subjecting each level to mutation for twenty times.

Figure 5.13 and Figure 5.14 shows the evolution of fitness across generations. The average fitness increases along the generation, while max fitness increases very slowly.

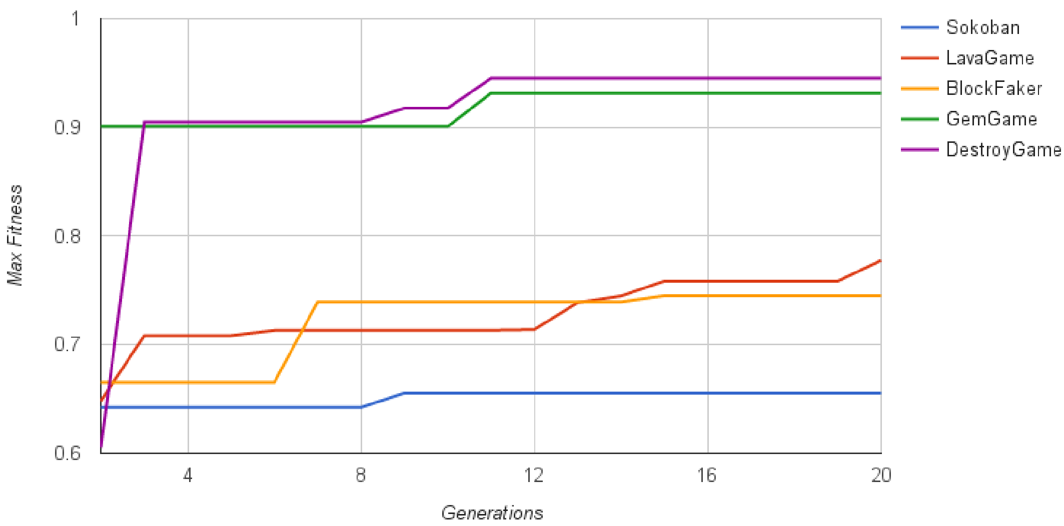

Figure 5.13: Maximum fitness for GA with mixed initialization

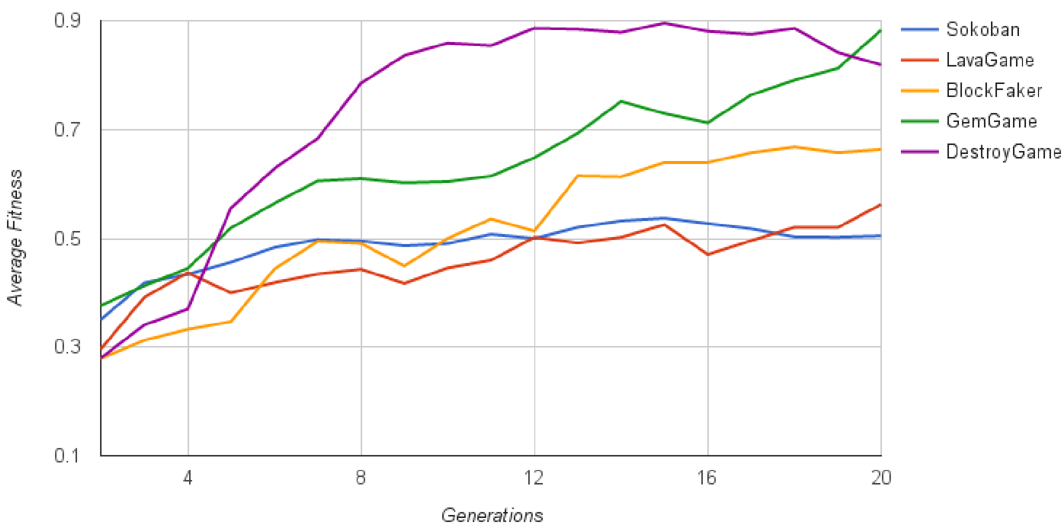

Figure 5.14: Average fitness for GA with mixed initialization

Figure 5.15 shows different generated levels using mixed initialization. All levels are playable with 42.5% repeated levels.

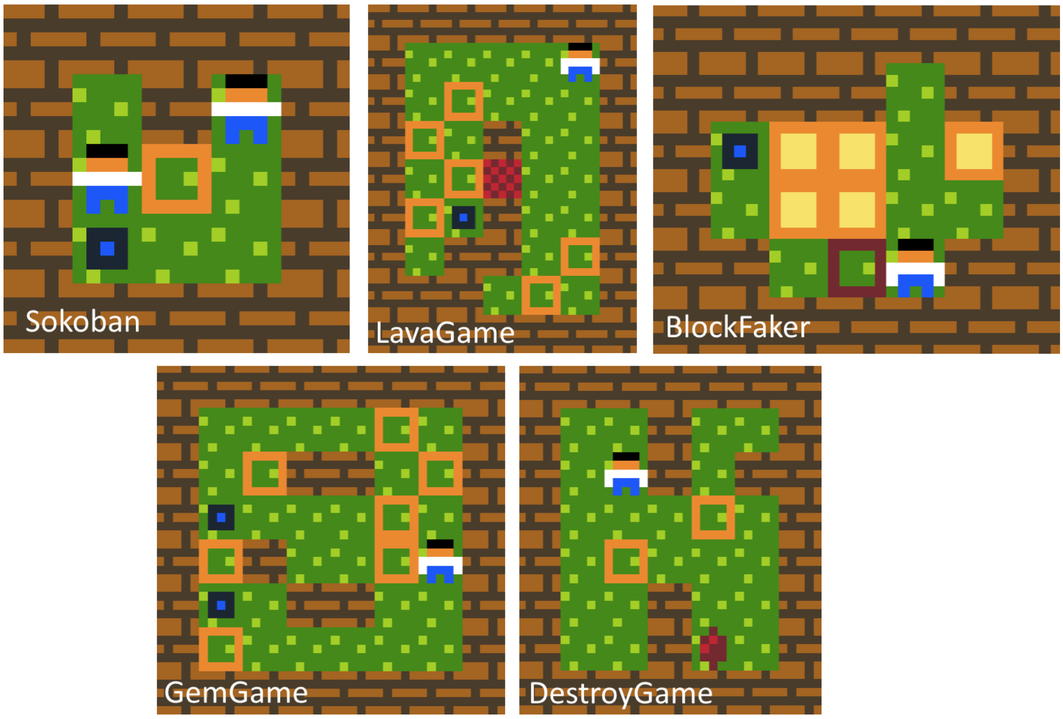

Figure 5.15: Examples of the generated levels using GA with mixed initialization

Sokoban		LavaGame		BlockFaker		GemGame		DestroyGame	
Automated Player	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**	**Automated Player**	**Human Player**
0.654089218	0.7	0.5989727	0.7	0.5658553	0.6	0.82325022	0.8	0.680589	0.4
0.654089218	0.7	0.5989727	0.7	0.5658553	0.6	0.82325022	0.8	0.680589	0.4
0.64291055	0.644444444	0.5544389	0.8	0.6580676	0.6	0.9621	0.8	0.9132655	0.8
0.64291055	0.64444444	0.5544389	0.8	0.6580676	0.6	0.9621	0.8	0.9132655	0.8
0.68702434	0.533333333	0.5526414	0.7	0.62174446	0.6	0.9430214	0.8	0.9396707	0.8
0.68702434	0.533333333	0.5526414	0.7	0.62174446	0.6	0.9430214	0.8	0.9390626	0.8
0.68636698	0.555555556	0.6992303	0.4	0.68210815	0.7	0.8828038	0.6	0.933516	0.6
0.6863669	0.555555556	0.6992303	0.4	0.68210815	0.7	0.8824508	0.6	0.933516	0.6
0.6894032	0.666666667	0.718745	0.8	0.7304408	0.8	0.924357	0.8	0.9286588	0.4
0.6894032	0.666666667	0.718745	0.8	0.7304408	0.8	0.924357	0.8	0.9286588	0.4
0.6994754	0.688888889	0.654869	0.7	0.7425093	0.9	0.931827	0.8	0.933017	1
0.6994754	0.688888889	0.654869	0.7	0.7425093	0.9	0.931827	0.8	0.933017	1
0.67699273	0.666666667	0.7309346	0.6	0.65543165	0.8	0.950839	0.6	0.9321156	0.8
0.67699273	0.666666667	0.7309346	0.6	0.65543165	0.8	0.950839	0.6	0.9321156	0.8
0.65561072	0.755555556	0.7776904	1	0.745198	0.8	0.93151312	0.8	0.9451245	0.6
0.65561072	0.755555556	0.7586242	1	0.745198	0.8	0.93151312	0.8	0.9451245	0.6
Correlation	-0.393092	**Correlation**	0.088979799	**Correlation**	0.814104796	**Correlation**	0.026758801	**Correlation**	0.511263461

Table 5.5: Automated Scores vs Human Scores for GA with mixed initialization

5.2.4 Different Techniques Comparison

5.3 Rule Generation

Table 5.6: Example table for demonstration

Chapter 6: Conclusion

In this research, the common industrial problem of …. As extension to this work, the following points are recommended for the future work;

References

[1] Akalabeth. `http://www.filfre.net/2011/12/akalabeth/`. [Accessed: 2015-01-18].

[2] Angry birds. `http://en.wikipedia.org/wiki/Angry_Birds`. [Accessed: 2015-03-17].

[3] Baghdadi, W., Shams Eddin, F., Al-Omari, R., Alhalawani, Z., Shaker, M., and Shaker, N. A procedural method for automatic generation of spelunky levels. In *Proceedings of EvoGames: Applications of Evolutionary Computation, Lecture Notes on Computer Science* (2015).

[4] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Computing Research Repository* (2012).

[5] Bfxr. `http://www.bfxr.net/`. [Accessed: 2015-01-19].

[6] Browne, C., and Maire, F. Evolutionary game design. In *IEEE Transactions on Computational Intelligence and AI in Games* (2010), IEEE, pp. 1–16.

[7] Chen, J. Flow in games (and everything else). *Commun. ACM* (2007), 31–34.

[8] Cook, M., and Colton, S. Multi-faceted evolution of simple arcade games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2011), IEEE, pp. 289–296.

[9] Cook, M., Colton, S., Raad, A., and Gow, J. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Applications of Evolutionary Computation - 16th European Conference* (2013), pp. 284–293.

[10] Cut the rope. `http://en.wikipedia.org/wiki/Cut_the_Rope`. [Accessed: 2015-03-17].

[11] Dahlskog, S., and Togelius, J. Patterns and procedural content generation: Revisiting mario in world 1 level 1. In *Proceedings of the First Workshop on Design Patterns in Games* (2012), ACM, pp. 1:1–1:8.

[12] Dahlskog, S., and Togelius, J. Patterns as objectives for level generation. In *Proceedings of the Workshop on Design Patterns in Games at FDG* (2013), ACM.

[13] Dahlskog, S., and Togelius, J. A multi-level level generator. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014).

[14] Dahlskog, S., and Togelius, J. Procedural content generation using patterns as objectives. In *Proceedings of EvoGames, part of EvoStar* (2014).

[15] Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T., and Togelius, J. Towards a Video Game Description Language. In *Artificial and Computational Intelligence in Games*, vol. 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 85–100.

[16] Every game studio that's closed down since 2006. http://kotaku.com/5876693/every-game-studio-thats-closed-down-since-2006. [Accessed: 2015-01-20].

[17] Font, J. M., Mahlmann, T., Manrique, D., and Togelius, J. Towards the automatic generation of card games through grammar-guided genetic programming. In *Proceedings of Foundations of Digital Games* (2013).

[18] Fruit dating. https://www.behance.net/gallery/13640411/Fruit-Dating-game. [Accessed: 2015-03-17].

[19] Generate everything. http://vimeo.com/92623463. [Accessed: 2015-01-19].

[20] Genetic algorithm. http://en.wikipedia.org/wiki/Genetic_algorithm. [Accessed: 2015-01-18].

[21] Gvg-ai. http://www.gvgai.net. [Accessed: 2015-04-04].

[22] Hausknecht, M., Khandelwal, P., Miikkulainen, R., and Stone, P. Hyperneat-ggp: A hyperneat-based atari general game player. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation* (2012), ACM, pp. 217–224.

[23] History of video games. http://en.wikipedia.org/wiki/History_of_video_games. [Accessed: 2015-01-18].

[24] How much does it cost to make a big video game? http://kotaku.com/how-much-does-it-cost-to-make-a-big-video-game-1501413649. [Accessed: 2015-01-20].

[25] Huebrix. http://www.huebrix.com/. [Accessed: 2015-03-17].

[26] Koster, R., and Wright, W. *A Theory of Fun for Game Design*. Paraglyph Press, 2004.

[27] Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., and Thompson, T. General Video Game Playing. In *Artificial and Computational Intelligence in Games*, vol. 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 77–83.

[28] Liapis, A., Yannakakis, G. N., and Togelius, J. Enhancements to constrained novelty search: Two-population novelty search for generating game content. In *GECCO '13 Proceedings of the fifteenth annual conference on Genetic and evolutionary computation conference* (2013), ACM, pp. 343–350.

[29] Lim, C.-U., and Harrell, D. F. An approach to general videogame evaluation and automatic generation using a description language. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014), IEEE, pp. 286–293.

[30] Lode runner. http://en.wikipedia.org/wiki/Lode_Runner. [Accessed: 2015-04-04].

[31] MURASE, Y., MATSUBARA, H., AND HIRAGA, Y. Automatic making of sokoban problems. In *PRICAI'96: Topics in Artificial Intelligence, 4th Pacific Rim International Conference on Artificial Intelligence, Cairns, Australia, August 26-30, 1996, Proceedings* (1996), pp. 592–600.

[32] NIELSEN, T. S., BARROS, G. A. B., TOGELIUS, J., AND NELSON, M. J. General video game evaluation using relative algorithm performance profiles. In *Proceedings of the 18th Conference on Applications of Evolutionary Computation* (2015).

[33] Overworld overview. http://bytten-studio.com/devlog/2014/09/08/overworld-overview-part-1/. [Accessed: 2015-03-15].

[34] PEREZ, D., SAMOTHRAKIS, S., AND LUCAS, S. M. Knowledge-based fast evolutionary MCTS for general video game playing. In *IEEE Conference on Computational Intelligence and Games* (2014), pp. 1–8.

[35] PEREZ, D., SAMOTHRAKIS, S., TOGELIUS, J., SCHAUL, T., LUCAS, S., COUETOUX, A., LEE, J., LIM, C., AND THOMPSON, T. The 2014 general video game playing competition. *IEEE Transactions on Computational Intelligence and AI in Games* (2015).

[36] PREUSS, M., LIAPIS, A., AND TOGELIUS, J. Searching for good and diverse game levels. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2014).

[37] Procedural generation of puzzle game levels. http://www.gamedev.net/page/resources/_/technical/game-programming/procedural-generation-of-puzzle-game-levels-r3862. [Accessed: 2015-02-24].

[38] Puzzle script. http://www.puzzlescript.net/. [Accessed: 2015-01-19].

[39] Random level generation in wasteland kings. http://www.vlambeer.com/2013/04/02/random-level-generation-in-wasteland-kings/. [Accessed: 2015-01-18].

[40] Rom check fail. http://www.farbs.org/romcheckfail.php. [Accessed: 2015-04-03].

[41] SCHMIDHUBER, J. Formal theory of creativity, fun, and intrinsic motivation (1990âĂŞ2010). *IEEE Transactions on Autonomous Mental Development 2* (2010), 230–247.

[42] SHAKER, M., SARHAN, M. H., NAAMEH, O. A., SHAKER, N., AND TOGELIUS, J. Automatic generation and analysis of physics-based puzzle games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2013), IEEE, pp. 1–8.

[43] SHAKER, M., SHAKER, N., TOGELIUS, J., AND ABOU ZLEIKHA, M. A progressive approach to content generation. In *EvoGames: Applications of Evolutionary Computation* (2015).

[44] Shaker, N., and Abou-Zleikha, M. Alone we can do so little, together we can do so much: A combinatorial approach for generating game content. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment* (2014).

[45] Shaker, N., Nicolau, M., Yannakakis, G. N., Togelius, J., and O'Neill, M. Evolving levels for super mario bros using grammatical evolution. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2012), IEEE, pp. 304–311.

[46] Shaker, N., Shaker, M., and Togelius, J. Evolving playable content for cut the rope through a simulation-based approach. In *Artificial Intelligence and Interactive Digital Entertainment* (2013), G. Sukthankar and I. Horswill, Eds., AAAI.

[47] Smith, A. M., Andersen, E., Mateas, M., and Popovic, Z. A case study of expressively constrainable level design automation tools for a puzzle game. In *Foundations of Digital Games* (2012), ACM, pp. 156–163.

[48] Smith, A. M., and Mateas, M. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games* (2010), IEEE, pp. 273–280.

[49] Smith, G., Treanor, M., Whitehead, J., and Mateas, M. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (2009), ACM, pp. 175–182.

[50] Snodgrass, S., and Ontanon, S. Experiments in map generation using markov chains. In *Proceedings of the International Conference on the Foundations of Digital Games* (2014).

[51] Sokoban. `http://en.wikipedia.org/wiki/Sokoban`. [Accessed: 2015-01-19].

[52] Sorenson, N., and Pasquier, P. The evolution of fun: Automatic level design through challenge modeling. *Proceedings of the First International Conference on Computational Creativity (ICCCX).* (2010), 258–267.

[53] Sorenson, N., and Pasquier, P. Towards a generic framework for automated video game level creation. Springer, pp. 131–140.

[54] Spelunky. `http://en.wikipedia.org/wiki/Spelunky`. [Accessed: 2015-03-15].

[55] Sweetser, P., Johnson, D. M., and Wyeth, P. Revisiting the gameflow model with detailed heuristics. *Journal : Creative Technologies* (2012).

[56] Sweetser, P., and Wyeth, P. Gameflow: A model for evaluating player enjoyment in games. *Comput. Entertain.* (2005), 3–3.

[57] Taylor, J., and Parberry, I. Procedural generation of Sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation* (2011), EUROSIS, pp. 5–12.

[58] The binding of isaac. `http://edmundmcmillen.blogspot.com/2011/09/binding-of-isaac-gameplay-explained.html`. [Accessed: 2015-03-15].

[59] The full spelunky on spelunky. `http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky`. [Accessed: 2015-03-15].

[60] The rip-offs and making our original game. `http://asherv.com/threes/threemails/`. [Accessed: 2015-01-21].

[61] TOGELIUS, J., AND SCHMIDHUBER, J. An experiment in automatic game design. In *IEEE Symposium on Computational Intelligence and Games* (2008), IEEE.

[62] TOGELIUS, J., SHAKER, N., AND NELSON, M. J. Rules and mechanics. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[63] TOGELIUS, J., SHAKER, N., AND NELSON, M. J. The search-based approach. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2015.

[64] TREANOR, M., SCHWEIZER, B., BOGOST, I., AND MATEAS, M. The micro-rhetorics of game-o-matic. In *Proceedings of the International Conference on the Foundations of Digital Games* (2012), ACM, pp. 18–25.

[65] Variation forever. `http://eis.ucsc.edu/VariationsForever`. [Accessed: 2015-04-04].

[66] What is the budget breakdown of aaa games? `http://www.quora.com/What-is-the-budget-breakdown-of-AAA-games`. [Accessed: 2015-01-21].

[67] Where's my water? `http://en.wikipedia.org/wiki/Where%27s_My_Water%3F`. [Accessed: 2015-03-17].

[68] WILLIAMS-KING, D., DENZINGER, J., AYCOCK, J., AND STEPHENSON, B. The gold standard: Automatically generating puzzle game levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2012).

[69] Yavalath. `http://www.boardgamegeek.com/boardgame/33767/yavalath`. [Accessed: 2015-04-03].