Automatic Puzzle Level Generation: A General Approach using a Description Language

Ahmed Khalifa and **Magda Fayek**
Computer Engineering Department
Faculty of Engineering, Cairo University
Cairo University Road, Giza, Egypt
amidos2002@hotmail.com, magdafayek@ieee.org

Abstract

In this paper, we present a general technique to evaluate and generate puzzle levels made by *Puzzle Script*, a videogame description language for scripting puzzle games, which was created by Stephen Lavelle(Puzzle Script). In this work, we propose a system to help in generating levels for Puzzle Script. Levels are generated without any restriction on the rules. Two different approaches are used with a trade off between speed (Constructive approach) and playability (Genetic approach). These two approaches use a level evaluator that calculates the scores of the generated levels based on their playability and challenge. The generated levels are assessed by human players statistically, and the results show that the constructive approach is capable of generating playable levels up to 90%, while genetic approach can reach up to 100%. The results also show a high correlation between the system scores and the human scores.

Introduction

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)(What is the budget breakdown of AAA games?); 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(How Much Does It Cost To Make A Big Video Game?).

That huge production cost is one of the reasons for the use of Procedural Content Generation(PCG). 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(Akalabeth). 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.

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(Generate Everything).

In this paper, we propose a system to generate playable puzzle levels without any restrictions or hacks. we utilize small prior knowledge about puzzle script to ensure playability and challenge.

Background

We can not generate general levels without having a methodology to describe the games. Video Game Description Language(VGDL) was originally invented to help on the work for General Video Game Playing(GVGP)(Levine et al. 2013) at Standford University. Puzzle Script(PS) is a VGDL created by Stephan Lavelle to help game designers and developers to create puzzle games(Puzzle Script). Games generated by PS are time stepped games similar to Sokoban(Sokoban). 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. In this work, we focus on rules, win conditions, and levels section. Rules are a set of production rules that govern how the game will be played. For example, *[> 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. Win conditions are group of rules that identify when the level should end. Levels are 2D matrices showing the current configuration for each game level using objects identified in objects section.

Literature Review

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.(Murase, Matsubara, and Hiraga 1996). Murase et al. work focused on generating well designed solvable levels for Sokoban(Sokoban). 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(Taylor and Parberry 2011) followed Yoshio Murase et al.(Murase, Matsubara, and Hiraga 1996) 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.(Murase, Matsubara, and Hiraga 1996) 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.(Murase, Matsubara, and Hiraga 1996) work. As the target number of crates increases, the generation process takes more time but delivers much more interesting and difficult levels.

Rychnovsky(Procedural Generation of Puzzle Game Levels) work focused on generating levels for his new game Fruit Dating(Fruit Dating). 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.

Shaker et al.(Shaker et al. 2013) worked on generating levels for physics based puzzle games. They applied their technique on Cut The Rope(CTR)(Cut the Rope). 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.(Shaker, Shaker, and Togelius 2013) 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.(Shaker et al. 2015) 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(Shaker, Shaker, and Togelius 2013). 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.

Williams-King et al.(Williams-King et al. 2012) work focused on generating levels for their game KGoldRunner. KGoldRunner is just a port for the famous game Lode Runner(Lode Runner).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.

Smith et al.(Smith et al. 2012) worked on generating puzzle levels for Refraction(Refraction). 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.

Methodology

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 **??**) 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 shows a high level block diagram of the system. The following subsections will describe each block in details.

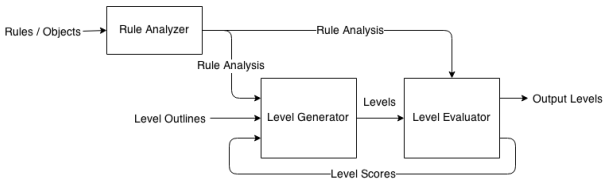

Figure 1: High level system block diagram for Level Generation

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.

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.

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.

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.

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

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 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 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 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: 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.

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

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;
```

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 **??**.

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 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.

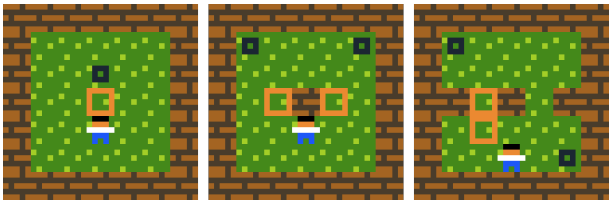

Figure 2: Examples on different levels for Sokoban game

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

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.(Lim and Harrell 2014) 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 **??**, 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. The left image in Figure 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. The right image in Figure 3 shows the same level from Sokoban, where the distance between player and winning objects (crates and targets) is highlighted.

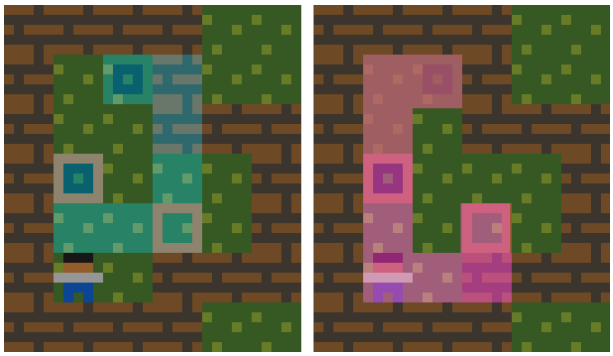

Figure 3: Example of distance between winning objects metric and distance between player and winning objects metric

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 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.

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] -> [|]

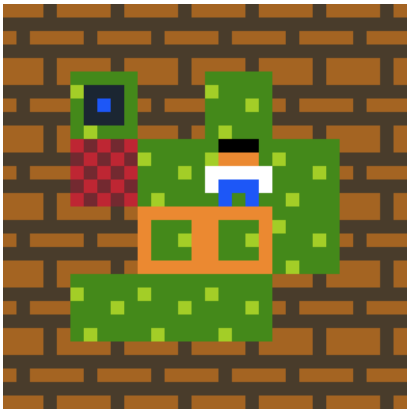

Figure 4: Example level from LavaGame showing the problem in the old metrics

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 **??**.

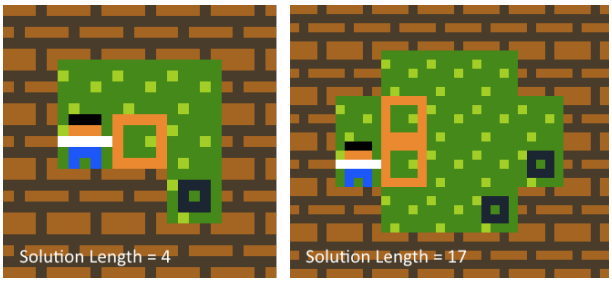

Figure 5: Examples of two sokoban levels with different area and solution lengths

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.(Nielsen et al. 2015) 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 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 shows different levels from Sokoban and their corresponding solution length. Its obvious that the seconds level have longer solution because it has bigger area.

  From the previous example we can conclude that the solution length depends on the level area. Instead of using the

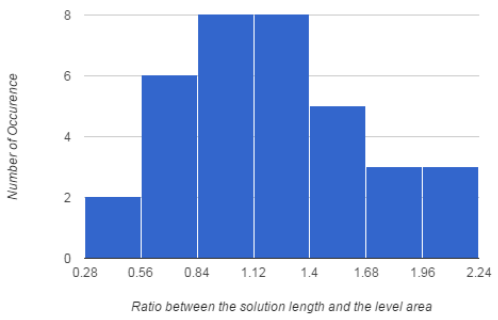

Figure 6: Histogram for the ratio between the solution length and the level area

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 .

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.

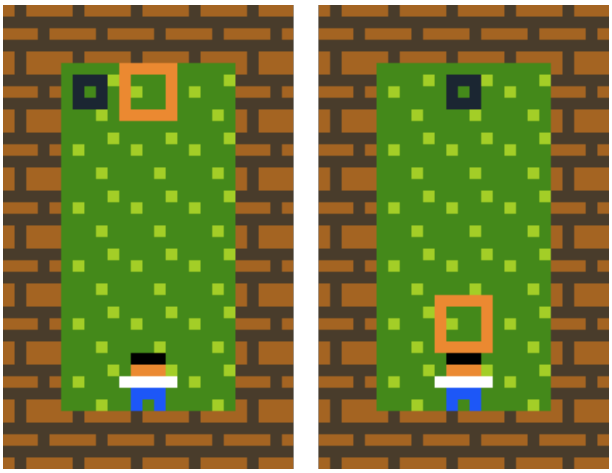

Figure 7: Example for two boring levels from Sokoban

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

  - *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(Taylor and Parberry 2011) 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 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.

To find the best ratio between both, We analyzed 40 hand crafted levels from 5 different games. A histogram is plot-

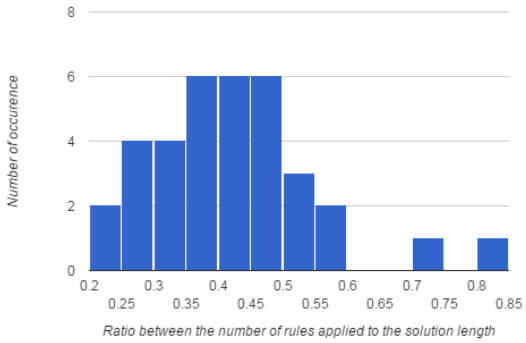

Figure 8: Histogram for the number of rules applied to the solution length

ted for the ratio between the number of applied rules and the solution length and shown in Figure .

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:

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

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

Results and Evaluation

Conclusion and Future Work

This research presented a system to generate levels and rules for Puzzle Script. Also, it proposed several metrics to evaluate puzzle levels and games based on their solution sequence.

The proposed system generates levels regardless of the game rules. It uses two different techniques (Constructive and Genetic approach). The constructive approach resulted in 90% playable levels which is enhanced in the genetic approach to reach 100%, but it needs more time. Genetic approach uses GA with three different initialization methods (Random initialization, Constructive initialization, and Mixed initialization). Random initialization produces levels with different configuration from the constructive approach, but with low playability equals to 75%. The constructive approach produces levels with playability reaching 100%, but with similar structure to the constructive approach. The mixed initialization is similar to constructive initialization in terms of playability but it expands the search space.

The generated levels are tested using human players[1] and a score is given for each level. Comparing the human scores with the system scores indicate a high correlation. This high correlation is a good indication that the proposed metrics can actually measure game playability and challenge. The correlation is higher in some games such as BlockFaker and Sokoban due to the high performance of the automated player in playing them.

This work is a first stone in general level and rule generation. There is a plenty to be done to expand and enhance it. As for future work, we aim to:

- analyze the effect of each metric on the level generation.
- utilize the metrics to analyze the search space for level generation.
- test different techniques rather than plan GA to increase the level diversity like in Sorenson and Pasquier work(Sorenson and Pasquier 2010).
- improve the time and the quality of the automated player to decrease the generation time.
- generate levels with a specific difficulty.

Acknowledgments

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 seen the light. Also Thanks to all my friends for the support and the huge help in collecting human feedback on the results.

References

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

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

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

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

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

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

[How Much Does It Cost To Make A Big Video Game?] 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-1561413649. [Accessed: 2015-01-20].

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

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

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

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

[Murase, Matsubara, and Hiraga 1996] Murase, Y.; Matsubara, H.; and Hiraga, Y. 1996. 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*, 592–600.

[Nielsen et al. 2015] Nielsen, T. S.; Barros, G. A. B.; Togelius, J.; and Nelson, M. J. 2015. General video game evaluation using relative algorithm performance profiles. In *Proceedings of the 18th Conference on Applications of Evolutionary Computation*.

[Procedural Generation of Puzzle Game Levels] 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].

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

[Refraction] Refraction. http://centerforgamescience.org/portfolio/refraction/. [Accessed: 2015-04-04].

[Shaker et al. 2013] Shaker, M.; Sarhan, M. H.; Naameh, O. A.; Shaker, N.; and Togelius, J. 2013. Automatic generation and analysis of physics-based puzzle games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

[Shaker et al. 2015] Shaker, M.; Shaker, N.; Togelius, J.; and Abou Zleikha, M. 2015. A progressive approach to content generation. In *EvoGames: Applications of Evolutionary Computation*.

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

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

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

[Sorenson and Pasquier 2010] Sorenson, N., and Pasquier, P. 2010. Towards a generic framework for automated video game level creation. 131–140. Springer.

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

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

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

[Williams-King et al. 2012] Williams-King, D.; Denzinger, J.; Aycock, J.; and Stephenson, B. 2012. The gold standard: Automatically generating puzzle game levels. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.