

Procedural Content Generation for Cooperative Games

Nuno Manuel Barbosa Lages Martins

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Rui Filipe Fernandes Prada
Member of the Committee: Prof. João Miguel De Sousa de Assis Dias

January 2021

Acknowledgments

Gostaria de agradecer ao Rui Prada e José Rocha pela orientação, ideias e ajuda que deram ao longo do desenvolvimento, especialmente neste ano particularmente mau para todos. Também gostaria de agradecer aos meus pais por me tolerarem estar tanto tempo em casa à frente do computador. Finalmente gostaria de agradecer aos meus amigos e colegas que me ajudaram a sobreviver a este longo e difícil ano.

Obrigado a todos.

Abstract

Procedural content generation is a popular topic in the games industry, it allows for faster development of content at reduced cost by being able to create infinite content. Creating levels and other details of the levels can reduce the workload on artists and game developers. Though a lot of work can still be done in procedural content generators, such as making generated content more diverse and realistic, when it comes to generating cooperative content, specifically content that requires collaboration between both players to be completed, there is a severe lack of work and approaches. We provide a solution to procedurally generating cooperative content. In this work, we create a level generator that uses a genetic algorithm as a base. We study how to properly define the problem and apply it to the game Geometry Friends as an example. We then evaluate our solution and finally we discuss how to keep improving the area for procedural content generation in cooperative games and propose different approaches.

Keywords

Procedural Content Generation, Cooperation, Genetic Algorithm, Geometry Friends

Resumo

A geração de conteúdo procedimental é um tópico popular na indústria de jogos, pois permite um desenvolvimento mais rápido de conteúdo a um custo reduzido, sendo capaz de criar conteúdo infinito. A criação de níveis e outros detalhes dos níveis pode reduzir a carga de trabalho dos artistas e desenvolvedores de jogos. Embora muito trabalho ainda possa ser feito em geradores de conteúdo procedural, como tornar o conteúdo gerado mais diverso e realista, quando se trata de gerar conteúdo cooperativo, especificamente conteúdo que requer cooperação entre ambos os jogadores para ser completo, há uma grave falta de trabalhos e abordagens. Oferecemos uma solução para gerar conteúdo cooperativo de maneira procedimental. Neste trabalho criamos um gerador de níveis que usa um algoritmo genético como base. Estudamos como definir corretamente o problema e aplicamo-lo ao jogo Geometry Friends como exemplo. Avaliamos a nossa solução e finalmente discutimos como continuar a melhorar a área de geração de conteúdo procedimental em jogos cooperativos e propomos diferentes abordagens.

Palavras Chave

Geração Procedimental Conteúdo, Cooperação, Algoritmo Genético, Geometry Friends

Contents

1	Introduction	1
1.1	Context	2
1.2	Motivation	2
1.3	Goal	3
1.4	Contributions	3
1.5	Outline	3
2	Related Work	5
2.1	Procedural Content Generation in Games	6
2.1.1	Content for Procedural Generation	8
2.2	Genetic Algorithms	11
2.2.1	Chromosome	12
2.2.2	Fitness Function	13
2.2.3	Selection	13
2.2.4	Crossover	15
2.2.5	Mutation	16
2.3	Cooperation in Games	17
2.3.1	Game Mechanics for Cooperative Games	18
2.3.2	Improving Cooperation in Games	19
2.3.3	Evaluating Cooperative Games	20
2.4	Procedural Content Generator for Cooperative Games	21
2.5	Geometry Friends	23
2.5.1	A level in Geometry Friends	24
3	Implementation	27
3.1	Software	28
3.2	Overview	28
3.3	Genetic Algorithm	29
3.3.1	Chromosomes	29

3.3.2	Fitness Function	30
3.3.3	Selection	38
3.3.4	Crossover	41
3.3.5	Mutation	45
3.3.6	Collectibles	47
4	Evaluation	49
4.1	Metrics	50
4.2	Experimenting	51
4.3	Level Generator	53
5	Conclusion	54
5.1	The Final Generator	55
5.2	Future Work	55

List of Figures

2.1	Example of vector and angle chromosomes	13
2.2	Example of single point crossover	15
2.3	Example of two point crossover	15
2.4	Example of mutations	16
2.5	Circle jump height	24
2.6	Rectangle climbing a ledge	24
2.7	Level in Geometry Friends	24
2.8	Level in with special platforms	24
2.9	Possible Sequence of moves to complete the level	25
2.10	Level divided into areas	25
3.1	The level generation from input to inside the game	29
3.2	First Chromosome genotype	30
3.3	Visual representation of input regions for the fitness function	31
3.4	Rafaels approach to calculate reachability	32
3.5	Rafaels approach with the improvement	32
3.6	Levels generated using the percentage relative to the entire level	36
3.7	Levels generated using 15% rectangle, 20% circle, 20% cooperative, 30% common as the input percentage relative to the entire level	36
3.8	Levels generated using the percentage relative to the reachable area	37
3.9	Levels generated using 10% rectangle 10% circle 40% cooperative and 40% common as the input percentage relative to the reachable areas	38
3.10	Visual representation of input regions used in selection method testing	39
3.11	Random selection	39
3.12	Stochastic selection	39
3.13	Stochastic selection without duplicate levels	40
3.14	Tournament with 4 participants	41

3.15 Tournament with 16 participants	41
3.16 Top 30% elitism	41
3.17 Top 30% elitism and maintaining the best	41
3.18 Level 1 before any changes	42
3.19 Level 2 before any changes	42
3.20 Levels representation as an array of integers	42
3.21 Children generated using a one point crossover with Levels 1 and 2 as parents	43
3.22 Children generated using a two point crossover with Levels 1 and 2 V2 as parents	43
3.23 Children generated using the specialized crossover with Levels 1 and 2 as parents	44
3.24 Two Children generated with Levels 2 and 1 v2 as parents using the algorithm 3.4	45
3.25 Levels 1 and 2 after uniform mutation	45
3.26 Levels 1 and 2 after a significant change mutation	46
3.27 Levels 1 and 2 after applying the SignificantMutateChromosome function	47
3.28 A level with collectibles placed and its representation in game	48
4.1 Time Taken per generation with population of 50	50
4.2 Time Taken per generation with population of 10	50
4.3 Time per evaluation of a level	51
4.4 GUI Tool to specify input, with some example input	52
4.5 Examples of input from testers and the final level generated	52

List of Tables

3.1	Example of levels Generated	35
3.2	Example of levels Generated	35

List of Algorithms

2.1	Layout of a Genetic algorithm	12
3.1	Sum of all intersections	31
3.2	Minimum of all intersection	33
3.3	Multiplication of all intersections	34
3.4	Function for creating a child from two parents	44
3.5	Function for significant mutations	46

1

Introduction

Contents

1.1	Context	2
1.2	Motivation	2
1.3	Goal	3
1.4	Contributions	3
1.5	Outline	3

1.1 Context

Games are a source of entertainment for many. They can provide a variety of experiences and help players learn or improve different skills. Multiplayer games are specially sought after as they allow players to join their friends in many challenges and promote friendship. Some by having them cooperate towards a common goal, other by having them compete against each other.

The tools for creating games keep improving and the game industry is larger than ever. The hardware to play games has also improved and allows for new experiences, such as virtual reality, or simply allows for larger scale games to be played and developed. But larger scale games take a lot of time to be developed and it is extra time consuming to design by hand every aspect of the game. For those reasons procedural content generation is used to help develop certain aspects of games including, the environment, by creating the trees, rocks or shaping the terrain, the characters that live in the game.

Rogue-like games are a genre where its main focus is that every time you play the game the world provides different challenges, Spelunky¹, Hades² and Noita³ are some examples. Spelunky and Hades gameplay focus on the player going through a series of rooms where as you progress their style and challenges change, they use PCG to create these levels, Noita on the other hand is more of an open-world game where it has clearly defined areas but you can go to any of those areas without a fixed order unlike the other examples, the combat system also uses PCG to create the wands each has a different spells and it is possible to combine wands, with so many possibilities they needed to be procedurally generated.

1.2 Motivation

Games that focus on providing cooperative challenges and puzzles that require two or more players can be very difficult to develop as they require the game designer to carefully design the level and test it to guarantee coherence between the different elements that create the cooperative challenges. For this reason procedural content generators for these challenges are hard to develop and those that exist allow multiple players but do not often need both players to solve the challenges generated. These generators tend to focus on generating a map that has different quests or enemies evenly spread out and that allows players to face them alone or with others making them easier with others.

Improving tools that allow procedural content generation is important as they also allow for smaller groups of designers to create larger worlds and they can reduce the costs of development.

While PCG has been subject to many studies, including studies that focused on the generation of

¹ Derek Yu 2008, Spelunky, video game, Microsoft Windos, Derek Yu.

²Supergiant Games 2018, Hades, video game, Supergiant Games

³Nolla Games 2019, Noita, video game, Nolla Games

cooperative challenges, there is still a lack of tools to help with designing cooperative focused levels.

1.3 Goal

The problem we are addressing is the procedural generation of cooperative levels, these are levels that focus on providing cooperative challenges that require cooperation to complete.

Our goal is to create a level generator that receives a series of requirements as input from the designer and generates a level that fulfills those requirements. To reach this goal we chose to use a genetic algorithm that can help us find the best solution, we chose a genetic algorithm as they allow us to spread the search for the solution and allows us to define a direct metric to evaluate the solutions. We intend for the input to be a description of where in the level should certain types of challenges be. We will test this solution by creating a level generator for the game Geometry Friends

1.4 Contributions

In this work our main contribution is an approach for creating a procedural cooperative level generator. We also present a collection of studies in the relevant areas, procedural content generation, cooperation in games, genetic algorithms, and we look at the game Geometry Friends and analyse it as a cooperative game. We then provide several heuristics, that receive input from a designer, and that are used in a genetic algorithm to create levels for the game Geometry Friends. We study the different selection, crossovers, and mutation methods and their effect in the level generation.

1.5 Outline

This document starts by examining some work related to areas of research connected to our problem. We begin with an overview of procedural content generation and their use in games, we start that by defining relevant properties in procedural content generator, followed by different methods on how to classify these generators. We then define what type of content can be created, and follow up with different methods that can be used. We then focus on genetic algorithms as they are the base for our generator. Next we write about cooperation in games, how it is defined, we show examples of cooperation, and then present different mechanics commonly used in cooperative games, we show to improve cooperative games, and then how to evaluate them. Next we discuss an example of an attempt at creating a procedural content generator specific to a simple cooperative 2d platform game. We finish this section with an overview of the game Geometry Friends. Finally we present our implementation, how we defined the parameters for the genetic algorithm and their different iteration. We end with an

evaluation of our implementation, how it relates to previously defined properties and present feedback from users testing our generator and the levels generated.

2

Related Work

Contents

2.1	Procedural Content Generation in Games	6
2.2	Genetic Algorithms	11
2.3	Cooperation in Games	17
2.4	Procedural Content Generator for Cooperative Games	21
2.5	Geometry Friends	23

2.1 Procedural Content Generation in Games

Procedural Content Generation(PCG) has been defined as the algorithmic creation of game content with limited or indirect user input [1], the game Civilization¹ uses PCG techniques to generate maps, No Mans Sky² uses it to create planets and its animals. Togelius et al [1] defined content as most of what is in a game, except things like the game engine and NPC AI (Non-playable Character Artificial Intelligence) behaviour, this definition is considered to be too broad so we will use a different one that is defined later.

When developing new tools it is important to be able to classify them, so some desirable properties for a PCG have been defined [1] as:

- **Speed:** PCG can be in real time, meaning while the player is playing, or can be generated before-hand such as during the development of the game and so can take months to generate or, if it is done during a loading screen right before playing, it can not be too slow, but does not have to be instant. Therefore the requirements for speed depend on how the content needs to be generated for the game.
- **Reliability:** This refers to whether the generator is completely random or if it can guarantee certain criteria, for example some generators might not need this quality but others may need to make sure that there is an exit to a dungeon otherwise it is not playable, on the other hand a texture that looks a bit weird is not game breaking.
- **Controllability:** If there is something that can be changed, or tweaked by a human to specify certain aspects, this is very important in player adaptive mechanisms.
- **Expressivity and diversity:** the ability to generate levels that are not just slight variations on the same thing, but also that are not all so different that they become senseless. This is one of the harder properties because of the difficulty of measuring what is expressive or diverse.
- **Creativity and believability:** this has to do with whether the generated content can be easily differentiated from human made content or not, mostly we want it to not be obvious.

The usefulness of these qualities obviously depend on the game, for example Minecraft³ requires the generation to be made in real time, as the player moves further away from the starting area the game will generate new environments, as opposed to Terraria⁴ where the map is generated once when the world is created, before the player enters the world. When looking at games where cooperation is taken into consideration when creating a level, real time speed can be a requirement that is difficult to meet

¹Firaxis, 2K Games 2011, Civilization VI, video game, Microsoft Windows, United States.

²Hello Games 2016, No Man's Sky, video game, Microsoft Windows, United Kingdom.

³Mojang 2009, Minecraft, video game, Microsoft Windows, Mojang, Sweden.

⁴Re-Logic 2011, Terraria, video game, Microsoft Windows, Re-Logic, Indiana, United States.

because there is a need for moments where players have to cooperate in order to progress. For puzzle games Reliability is a necessity, for open world games there can be less of it, for example in Spelunky⁵ there is a need to guarantee an exit in the each level. Left 4 Dead⁶ is a game that experimented with player adaptive mechanisms in a procedural way, they look at the emotional intensity of players and if it is high they remove the generation of threats for some time. Minecraft and Terraria are capable of high level of expressivity with many different maps. Left 4 Dead changes the placement of weapons and the spawn of monsters, this requires a certain amount of believability because it does not make sense to have a monster spawn on top of a building where it can not interact with the player, or weapons appear in non-reachable areas.

Those properties are not enough to compare different generators so Togelius et al [1] also defined a taxonomy of PCG that consists of the following dimensions:

- **Online vs Offline:** online is the generation of content as the player is playing the game while offline is generating during the development of the game or before the player starts a game session.
- **Necessary vs Optional:** Necessary content is by definition content that is required for completion of a level, while optional is not. This mostly affects the fact that Necessary content needs to be correct, meaning a level must be possible to be completed after generation, while Optional does not.
- **Degree and Dimension of control:** this refers to how the humans can control the generated space, if a seed is used, then using the same seed will generate the exact same level, or a set of parameters can be used to have better control on the generation of the content over several dimensions.
- **Generic vs Adaptive:** Generic methods do not take the players behaviour into account while Adaptive will analyse the behaviour and determine a set of parameters to adapt the generation of content.
- **Stochastic vs Deterministic:** Stochastic means that the content generated can not be exactly regenerated, due to the randomness in the generation, as opposed to Deterministic where given the same input it will generate the same output.
- **Constructive vs Generate-and-test:** in constructive the generation of the level occurs in one pass while in Generate-and-test it is an iterative process where first a level is generated, then it is tested and then we make adaptations and generate again, until a satisfactory solution is reached.

⁵Derek Yu 2008, Spelunky, video game, Microsoft Windows, Derek Yu.

⁶Turtle Rock Studios 2008, Left 4 Dead, video game, Microsoft Windows, Valve, United States.

- **Automatic generation versus mixed authorship:** Automatic allows for limited input from the game designer, making possible to tweak some parameters. Mixed authorship tries to allow the designer to give a more abstract input for example drawing part of a 2D level and having the algorithm generate the rest based on constraint satisfaction.

When applying these dimensions to puzzle games, we can see that, normally puzzle games have the entire puzzle created beforehand so the content is not often generated while the player is playing. Since a puzzle game needs to have a solution the generator has to guarantee the existence of a solution, and since we hope to apply these in cooperative games then the generator should also guarantee the necessity for cooperation. The degree and dimension of control is very dependant on which type of algorithm is used, some allow more or less control and this algorithm will also affect how stochastic or deterministic the generation process is. We explore the different types of algorithms later. If it is possible to define good metrics, that can be tested automatically, then a Generate-and-test approach can help get better levels, possibly ones that have more opportunities for cooperation, however a Constructive approach is just as valid, since in puzzle games the levels can be created beforehand and so the designer can make changes. Since cooperation is a difficult concept to define in an algorithm or through metrics a mixed authorship approach might be able to provide better results. However good results have been achieved with a fully constructive approach [2].

2.1.1 Content for Procedural Generation

The definition of content by Togelius et al [1] is too broad, Hendrikx et al [3] defined content that can be procedurally generated by separating it into layers based on how the content can be created from other content, these layers are: game bits, game space, game systems, game scenarios, game design, and derived content.

Game Bits are the lowest layer and therefore the most elementary units of game content, typically not used to engage the user when considered independently. They can be Concrete or Abstract, Concrete is content that can be interacted with such as trees and items, Abstract is content that needs to be combined to create concrete content such as textures and sound.

Some examples of game bits are:

- **Textures (Abstract):** Images that add detail to geometry and models or visual representation of menus
- **Sound (Abstract):** Music is used to set the atmosphere and pace of the game while general sound effects are used as feedback to the player.
- **Vegetation (Concrete):** Adds realism to the game making it more immersive.

- **Buildings (Concrete):** Essential for representing urban environments and special buildings often affect the players decision on where to go.
- **Behavior (Concrete):** Describes the way in which objects interact with each other and the environment. This helps the game be more interesting.
- **Fire, Water, Stone and Clouds (Concrete):** Help create more believable worlds.

Game Space is the environment in which the game takes place, this is partially filled with game bits in which the player is going to navigate. It can also be considered Concrete or Abstract, Concrete spaces tend to closely resemble spaces the way humans perceive them, Abstract is more an imagined space for example the board in the game of chess.

Examples of game spaces are:

- **Indoor Maps (Abstract or Concrete):** Structures and relative position of space partitioned into rooms, that can be connected by corridors, or overlapping and connected by stairs. Can also be caves with varying and unusual geometry.
- **Outdoor Maps (Abstract or Concrete):** Can be depictions of elevation and structures of outdoor terrains, normally outdoor maps tend to then have indoor maps.
- **Bodies of Water (Concrete):** Can be rivers, lakes and seas and are often used as obstacles or sometimes interactive game space.
- **Other map features (abstract or concrete):** such as teleportation areas and mountains, ridges, ravines, may also be part of game space.

Game Systems is a representation of the relation between content in the game, it can be seen as Abstract for example the relation between Vegetation and its relation with the features of the outdoor maps, or Concrete like cities and networks of cities

Some Examples are:

- **Ecosystems (Abstract or Concrete):** Describes the placing, evolution and interaction of flora and fauna.
- **Road Networks (Abstract or Concrete):** form the basic structures of an outdoors map, used for transportation between points of interest
- **Urban Environments (Abstract or Concrete):** large clusters of buildings where many people live and interact with their surroundings.
- **Entity Behaviour (Concrete):** Very important for making the player experience in the virtual world life-like. This is done by defining the relation between NPCs and the player including their possible

interactions. Hendrikx believes that not only does player interaction requires complex entity behavior, but also that group movement patterns are examples where procedural algorithms could achieve more realistic results.

Game Scenarios describe how the game unfolds. They can describe it in an Abstract way by describing how other objects inter-relate, in a Concrete way by explicitly showing it to the player as part of the game narrative.

Some Examples are:

- **Puzzles (Abstract):** Are problems that use previous knowledge to be solved or there is a limited amount of possible solutions in which case they can be solved by exploring that finite space i.e. attempting every possible solution.
- **Storyboards (Abstract or Concrete):** Are sequential panels describing a scene or event.
- **Story (Abstract or Concrete):** Presents the background for the events and goals that the player will go through
- **Levels (Abstract or Concrete):** Are normally used as separators between gameplay sequences. They are made by grouping the previous elements and consist of a playable environment that has its objectives, such as solving a sequence of puzzles, or collecting some object.

Game Design can refer to all previous types of content including game design itself, it can be seen as rules or goals, aesthetic components, story or themes.

Part of the game design is:

- **System Design(Abstract):** are the patterns and rules underlying a game.
- **World Design (usually Concrete):** is the appearance of the world, when and where the story is taking place and what is the story. [4]

Derived Content are things that are a byproduct of the game world. This content helps the player immerse themselves by allowing the player to record their in-game experiences for review inside or outside the game.

For Example:

- **News and Broadcasts (Concrete):** By showing the player or their actions in the news in the game's universe.
- **Leaderboards (Abstract):** players ranking tables.

There are many different ways of creating a Procedural Content Generator, Hendrikx et al [3] compiled a collection of them and related them to which type of content they could be applied to. First they

created a taxonomy to classify the different types of algorithms based on the main basis of the algorithm, this taxonomy identified six groups: Pseudo-Random Number Generators (PRNG), Generative Grammars (GG), Image Filtering (IF), Spatial Algorithms (SA), Modelling and Simulation of Complex Systems (CS), Artificial Intelligence (AI).

Pseudo Random Number Generators(PRNG) are algorithms that generate a sequence of numbers that resemble a sequence of random numbers.

Generative Grammars(GG) consider grammar as a system of rules that through combination of them forms grammatical sentences in a given language, by considering objects encoded as letters or words they can be used generate structures.

Image Filtering(IF) algorithms try to improve an image in regards to a (subjective) measure, or to emphasize certain characteristics, such as displaying (partially) hidden information.

Spatial Algorithms(SA) manipulate space to generate game content, they receive a structure like a grid or a recursive functions.

Modeling and Simulation of Complex Systems(CS) is used to overcome the difficulties that exist when describing natural phenomena with mathematical equations. They sometimes focus on specifying local interactions and then look for emerging behaviour.

Artificial Intelligence(AI) is a field that tries to mimic animal or human intelligence. Examples are speech recognition, planning, execution of tasks by robots, and search of solutions for optimization problems.

We looked at what each area had and how they could help in our problem and decided that an AI to search for levels that fit our requirements could be the solution. We decided on a genetic algorithm as they have been successful when applied to complex problems such as Bin packaging [5] and timetable scheduling problems [6]. They have also been used in generating content for games, Moghadam et al [7] used it define the rhythm and difficulty of an endless runner. Connor et al [8] used a genetic algorithm to create a game level, it would generate the map and rate it the ratio of space that was traversable and whether there was a path from the beginning to the end. Genetic algorithms have also been used to generate levels for 2d platforming games, Mourato et al [9] applied a genetic algorithm to generating levels for the 1989 Prince of Persia⁷. Given these varied uses of the genetic algorithms we believed it could generate good results for our problem as well.

2.2 Genetic Algorithms

Genetic algorithms fall into the Artificial Intelligence group of algorithms, according to Hendrikx et al [3]. This is because the idea behind them is to mimic biological evolution. Goldberg et al [10] defines them

⁷Jordan Mechner, Brøderbund Software, 1989, Prince of Persia, MS-DOS, Brøderbund Software, Ubisoft

Algorithm 2.1: Layout of a Genetic algorithm

```
Function GeneticAlgorithm(populationSize):  
    population  $\leftarrow$  initialize(chromosome.create, populationSize)  
    CalculateFitness(population)  
    while not StopConditon(population) do  
        Selection(population)  
        Crossover(population)  
        Mutation(population)  
        CalculateFitness(population)  
    return population
```

as 'search algorithms based on the mechanics of natural selection and natural genetics' and they are used to solve optimization problems. They mimic biological evolution because they define a population that is described by structures that are similar to chromosomes, then they evolve that population through reproduction and some suffer mutations.

Genetic algorithms 2.1 can be divided into these parts: the chromosome or individual, the fitness function, the selection method, the crossover method and the mutation method. The stop condition is generally until the population converges or a maximum number of generations.

2.2.1 Chromosome

The chromosome represents an individual in a population and they are considered as a possible solution to the problem. They are a sequence of genes, a set of parameters or variables that represent our solution, also called the genotype, which is the encoding of the chromosome, i.e. its representation. Usually they are represented by a string of bits, binary values that are either 1 or 0, but the representation varies a lot depending on the problem in question. From these genotypes comes the phenotype which is where we get the expression or meaning in the sequence of genes, what does each part of our chromosome represents in our problem. It is possible that the chromosome representation does not have a fixed size.

For example, we are trying to determine a path from one point to another in a 2 dimensional plane and for this we use a chromosome that represents a series of steps, it can have a fixed number of steps or it can have a minimum and maximum number, we can then use an array of 2D vectors or an array of integers to store our chromosome, these are the genotypes. When we determine what those vectors then represent is when we are defining the phenotype, for example, each vector could represent a direction in which to move at every step, or each integer could represent an angle amount that determines how much it should turn at that step and then it would move according to each step for a predetermined amount of time. We can see these representations then vary a lot when applied, in the vector representation at each step its path is only determined by that step's vector, while the angle is

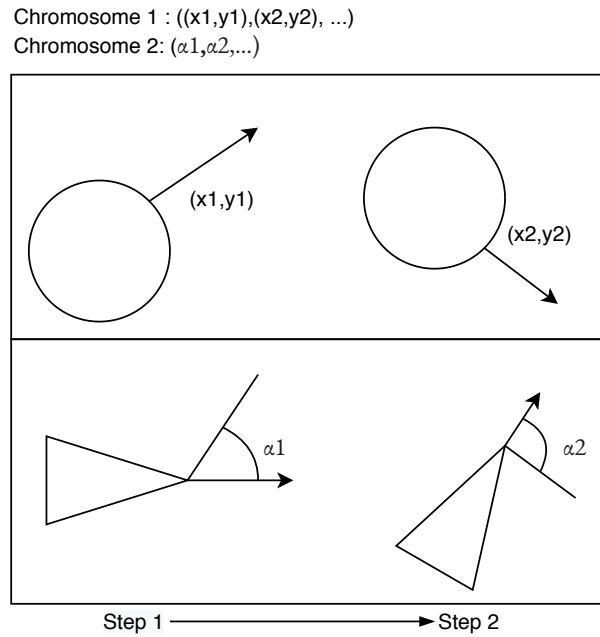


Figure 2.1: Example of vector and angle chromosomes

influenced by each previous step.

2.2.2 Fitness Function

The fitness function is where the problem we are trying to solve is defined and where we interpret the chromosome. Its objective is to give a value to each individual in the population, normally the values are between 0 and 1 where 1 is the best and is considered to be a solution to the problem. This function is where most of the effort in a genetic algorithm should be applied. It is important to properly define a function that approximates the designer's goal. If improperly defined the function might converge to a bad solution, that is one that the designer does not actually want, or might not converge at all.

It is common to iteratively define a fitness function, because it might not result in what is functionally desired or, sometimes it might become too computationally expensive, so experimentation is required to find a balance between the functional requirements and the performance requirements. When it is too computationally expensive, using a fitness approximation method (which takes information from previous evaluations to determine the new one) is a possible way to optimize the fitness function.

2.2.3 Selection

The selection is done to choose which individuals from the population should be used for creating the next generation through crossover. The main goal of this process is to help the algorithm converge.

This is because in general the selection process chooses the best chromosomes, that is the ones with highest fitness.

There are many approaches to the selection method. Some of these approaches are different types of fitness proportionate selections, that is that they use the fitness of each individual and their overall relation to the population to determine the likelihood that they are chosen. Examples of these selections methods are the Roulette Wheel, Rank, and Stochastic Universal Sampling selection.

The Roulette Wheel [11] takes all of the fitness values of the population and puts them in an array. It then normalizes it like a vector, afterwards it creates an array with the cumulative percentages, which is calculated by going through the normalized vector and at each index it adds the fitness values of all the previous indexes and inserts it to the new array. This is similar to creating a wheel where the slices for each member is proportional to their fitness. Afterwards it then spins that wheel by generating a random value x between 0 and 1 and searching in the array for the first index whose value is greater than x . Then the chromosome related to that index is chosen as a parent. This is repeated for as many parents are necessary and it is possible that the same chromosome is chosen more than once.

The Rank selection functions much like this but it first sorts the population by fitness and then assigns a new fitness value from 1 to n where 1 is given to the worst chromosome and n is given to the best, it then applies the roulette wheel method with the new fitness values, again the same chromosome can be chosen several times.

The Stochastic Universal Sampling is equal to the roulette wheel except when choosing from the array, instead of generating a new number for each choice it generates a number only once and then increases it by a fixed amount, this step amount is calculated as $1/n$, where n is the number of parents to be selected. This approach increases the chances that chromosomes with lower fitness will be selected, because when a member has a high fitness value in comparison to the other members it can saturate the wheel with a larger slice. By using steps it can better guarantee that it will eventually not choose that higher fitness chromosome. This is important because while the selection method wants to help the algorithm converge it still needs to not have it converge too early.

The Tournament Selection is a method that runs a small tournament with random individuals of the population for each choice, the winner of the tournament is the member with the highest fitness. The tournament size determines the likelihood that weaker individuals get selected, for example a tournament size of 1 would be the same as choosing randomly, while a size of 2, which is normally the default size, would guarantee that the lowest fitness is never chosen and would on average be better than one, however Lavinias et al [12] experimented with the tournament size and found that the most common sizes, 2 and 3, are not always a good choice and the size should be adjusted to the problem. Very large tournament sizes can make it so that the fittest chromosomes are chosen more commonly, and since normally the tournament selection allows for the same chromosome to participate in several tour-

naments, it is possible that, with a large tournament size, the selected individuals will not be very varied and diverse.

Elitism selection method sorts the chromosomes by fitness value and chooses the k chromosomes with the highest fitness, if k is equal to the size of the population it chooses the entire population, this selection method does not choose the same chromosome twice, but if you want to maintain the size of the population you would still choose a k smaller than the population, which would require you to use the same chromosome as parent more than once. There is a version of elitism that selects parents for the crossover but also holds on to the previously selected parents for the next generation, but they will still be affected by the mutation step.

2.2.4 Crossover

The previously selected chromosomes will act as parents and mate in order to cross their genes and create the children or individuals that compose the next population. The reason for the crossover is to try and diversify the population while passing on features from the parents to the children. There are many different ways for the parents to crossover, among the most known are the single-point and the two-point crossover.

The single point crossover randomly chooses an index in the sequence of genes and the first child would get every gene before that point from the first parent and the second child would get every gene before that point from the second parent, then the first child would get the bits after that point from the second parent and vice versa for the second child. The two point crossover would do the same but would choose two indexes, as shown in figures 2.2 and 2.3. There is also a K-point crossover variant that chooses k points, so with 1 or 2 as k , it would mimic these crossovers.

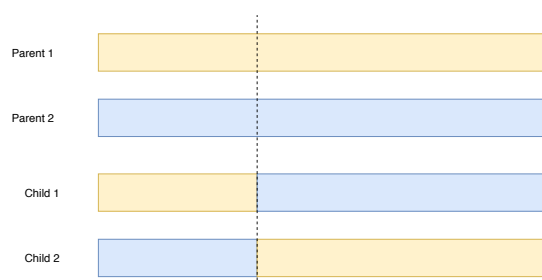


Figure 2.2: Example of single point crossover

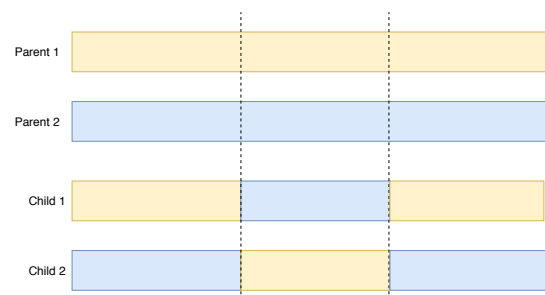


Figure 2.3: Example of two point crossover

There are versions of these crossovers that do not guarantee that the size remains the same for the chromosome, normally considered to be the messy versions, these can be used for chromosomes that do not have a fixed size.

Another crossover method is the uniform crossover, this method creates offspring where each gene

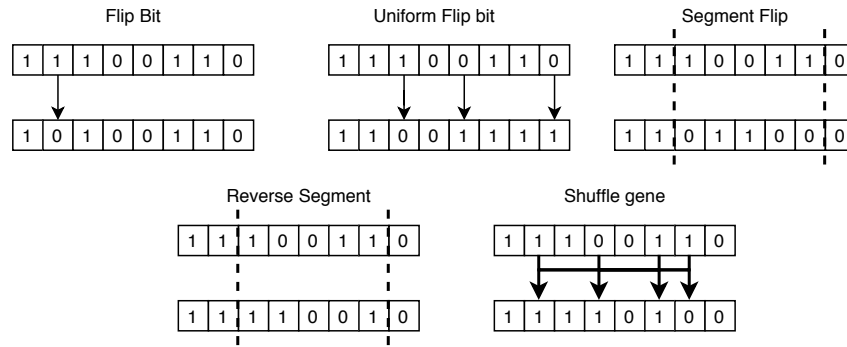


Figure 2.4: Example of mutations

is chosen with a certain probability, normally an equal probability, which parent it inherits from.

Typically in crossover methods two children are generated, where one has elements from both parents and the other has the remaining elements from the same parents, this is normally done so that all feature from both parents are still somewhere in the population. This is not a necessary requirement for crossover methods. With k point crossover, if only one child was created, the beginning of the second parent is not passed on, with the uniform crossover this might not happen since both chromosome, normally, have equal chances of being chosen.

Something that can be necessary to define a crossover specific to your chromosome, this can normally happen if there are certain guarantees that need to be fulfilled when creating the offspring. An example of this are some problems where the chromosome represent a list of items where no item repeats itself and every chromosome has the same list in a different order, these are used for order based problems (e.g.: the travelling salesman). Since this is a common problem some crossovers specific to these problems have already been developed such as the cycle crossover [13], partially-mapped crossover(PMX) [14], the uniform partially-mapped crossover (UPMX) [15], and ordered crossover(OX1) [10].

2.2.5 Mutation

The main purpose of the mutation is to help diversify the population and stop it from converging early, it does this by altering the chromosome in different ways: it can alter the gene values or it can switch the order of the genes. Not every individual is affected by a mutation, this is controlled by a mutation probability that should not be set too high or else the search has a risk of becoming too random. However it might be better might be better to have a lot of diversity in the first generations of a genetic algorithm and, as higher fitness solutions are found, the mutation probability can be decreased.

The flip bit mutation is a classic mutation method. This requires the chromosome to be a sequence of bits and then chooses a random bit and changes its value, from 1 to 0 and vice versa, this operator

can also be seen as a not function. Other variations on this might choose a segment of the chromosome and flip those, or they can go through the genes in a uniform manner and, with a random chance, flip each gene. These methods all have non bit versions that instead of choosing an opposite value they generate a new random value to replace the gene. Other methods involve switching the order, some reverse the gene order, others reverse a segment, some shuffle the index of the genes.

2.3 Cooperation in Games

Cooperation is a concept that has existed for millennia, it is the process of a group working together towards a common goal. Humans enjoy cooperating together so bringing it to video games was expect. Cooperative video games are believed to have started in 1978 with *Fire Truck* arcade game developed by Atari where two players would cooperatively steer the vehicle. Later with the introduction of consoles players would be able to use of multiple controllers, then with the origin of the internet and online multiplayer came the surge of cooperative games like MMORPGs. With the internet there came a need to differentiate between local co-op, where the game is designed for players that are using the same display screen, and online co-op, where each player would use their own screen.

Zagal et al [16] refers to three types of games categories in game theory: Competitive, Cooperative and Collaborative. Although originally only competitive and cooperative where considered, Marschak et al [17] refers to how collaboration in a team differs from cooperation. In competitive games, players have goals that oppose each other, therefore they need to create strategies that oppose the strategy from the other player, so this types of games are not relevant for our goal.

In Cooperative games both players have different goals but they do not necessarily oppose each other, this means that they may want to help each other in order to get a better results for themselves. These two definitions were part of the traditional Game Theory.

Later came the third category Collaborative games. In these games all participants are in a team therefore they all share the outcomes. While participants may have different knowledge and abilities, by sharing the same goal and outcomes they will all have to work together to maximize their utility. So they differ from cooperative games where players are not forced to cooperate to reach their different goals. In collaborative games players have the same goal and therefore need cooperation in order to get the best outcome.

We will not be making this distinction because in general when we talk about cooperation in video games we do not differentiate between that and collaboration, others authors, that we will be mentioning ahead [18] [19], also do not make that difference. Therefore whenever there is an opportunity for players to work together we consider that as cooperation, whether they have the same or different end goals.

For example in Dead By Daylight⁸ the player is part of a team of four that are trying to get away from a monster. The player wins by escaping, to that they need to open the door to escape, before opening it all generators around the map need to be fixed. It is cooperative because some players can distract the monster while others fix the generators, players can save others that have been caught by the monster, and so they need to cooperate, trying to win alone is much harder. However each players end goal is to escape, the others can be caught, as long as they escape, they win.

On the other hand we can look at Overcooked⁹, or the multiplayer mode from Portal 2¹⁰. In these games the rewards are shared among the players and so the player wins if the other players also win. However these two are still different types of cooperation, in Overcooked, depending on if the players are completely separated from each other, one player can be enough to finish the level while in Portal it needs both players to complete the level. We consider all these games as cooperative games.

2.3.1 Game Mechanics for Cooperative Games

Cooperation in video games comes in many different ways and Rocha et al [18] defined several common Design Patterns and Challenges Archetypes that appear in video games. These design patterns are:

- **Complementarity:** the most common design pattern, it tries to make sure characters complement each other, by having different abilities or different roles.
- **Synergies between abilities:** Guarantee that the abilities of one character synergize with the abilities of another character.
- **Abilities that can only be used on another player:** this type of abilities is to incentivize cooperation.
- **Shared Goals:** simple design pattern to make player work together, by giving groups of players goals that are not restricted to one player and therefore can be completed in a group.
- **Synergies between goals (Interlaced/Intertwined goals):** even if players have different goals one approach is to create some sort of synergy between the goals in order for them to cooperate.
- **Special Rules for Players of the same Team:** it is possible that the same action has different outcomes if done on a team member versus when it is done on an opponent. This can facilitate cooperation.

These were later extended by Magy et al [19] where they added:

⁸Behaviour Interactive 2016, Dead By Daylight, video game, Microsoft Windows, Montreal, Canada

⁹Ghost Town Games 2016, Overcooked!, video game, Microsoft Windows, Cambridge, United Kingdom

¹⁰Valve 2011, Portal 2, video game, Microsoft Windows, Bellevue, Washington, US

- **Camera Setting:** There are three main ways to have a shared screen in games, split horizontally or vertically, one character in focus, all characters in focus (the screen moves when all characters are near each other).
- **Interacting with the same object:** providing objects that can be manipulated by abilities from both characters.
- **Shared Puzzles:** to give players a shared challenge of obstacle.
- **Shared Characters:** providing a character that both players can use but only one at a time, making them discuss how to share the character.
- **Special characters targeting lone wolf:** having an enemy or obstacle that makes it much harder for a player to work alone, for example an enemy that grapples a player.
- **Vocalization:** having player characters inside the game give feedback of nearby dangers or points of interest.
- **Limited Resources:** limiting resources so that players have to share and exchange their own resources.

Reuter et al [20] and Hullettand and Whitehead [21] also use patterns for describing cooperation, but instead of the patterns describing game mechanics they describe gameplay sections, for example a pattern that describes a segment of the level as: the players have to both interact with two different buttons at the same time, this pattern could then be called Timed Two Man Rule¹¹. This is useful because by defining them this way we can look at a level as a sequence of these patterns.

2.3.2 Improving Cooperation in Games

Zagal et al [16] explored cooperation in board games, however they decided to use the term collaboration instead because they believed that using game theory's definition then cooperative games would still look for only one winner, in which case cooperative games could promote anti-collaborative practices, such as free riding, when a player does not work as much but still benefits, and backstabbing, which is when another player cooperates with another but then that other player defects allowing them to gain an advantage.

For this study they looked specifically at the Lord Of The Rings game by Reiner Knizia. This board game is about getting the hobbit that is carrying the ring from one side of the board to the other before they are corrupted and caught by Sauron, when a player is caught they are removed from the game. The players draw tiles and play cards to advance on the board, they then can choose to draw cards or reduce

¹¹ The two-man rule is a control mechanism that to give access to actions it requires the presence of two people.

their state of corruption, if a player plays their last card they increase their state of corruption. Analysing this game they concluded four lessons and three pitfalls that they extended to other cooperative games.

Lesson 1: “To highlight problems of competitiveness, a collaborative game should introduce a tension between perceived individual utility and team utility.”

This lessons tries to address selfish behaviour that affects the team. For example a player draws a tile where he can choose between increasing his corruption by 2 or increasing every players corruption by 1. We can look at individual utility as either increasing corruption by 2 or 1, but when looking at team utility in a team of 5 players it becomes increasing corruption by 2 or 5. A selfish player will choose to have his individual corruption be increased by 1 while the teams corruption is increased by 5.

Lesson 2: “To further highlight problems of competitiveness, individual players should be allowed to make decisions and take actions without the consent of the team.”

Lesson 3: “Players must be able to trace payoffs back to their decisions.”

Lesson 4: “To encourage team members to make selfless decisions, a collaborative game should bestow different abilities or responsibilities upon the players.”

Although these lessons can be used when designing a cooperative level generator they are more centered in game mechanics and are more useful for designing a cooperative game in itself.

Pitfall 1: “To avoid the game degenerating into one player making the decisions for the team, collaborative games have to provide a sufficient rationale for collaboration.”

Pitfall 2: “For a game to be engaging, players need to care about the outcome and that outcome should have a satisfying result.”

This pitfall obviously applies to all games, but more so to cooperative games, if players are not motivated they will not care enough to help each other, and if the outcome is boring or unrelated to the actions, i.e. a random outcome, then they wont want to learn the consequences of their actions.

Pitfall 3: “For a collaborative game to be enjoyable multiple times, the experience needs to be different each time and the presented challenge needs to evolve.”

The first pitfall is in a way related to how information is portrayed in a game, this is important to take into consideration because a level is a visual representation of information, so a generator has to take into consideration how to show the level, beyond making it playable. The third pitfall is the one we are directly addressing by creating a procedural content generator.

2.3.3 Evaluating Cooperative Games

Magy et al [19] defined Cooperative Performance Metrics(CPM), these are associated with observable events, they used them to analyse play sessions for different cooperative games, such as Rock Band 2, Lego Star Wars, Kameo, and Little Big Planet. The final set of CPMs they developed where:

- **Laughter or excitement together**, which is associated to events where players laughed due to a game event, or verbally expressed their enjoyment, or their facial expressions showed happiness or excitement.
- **Worked out strategies**, this is associated to talking about how to solve a challenge or how to approach a challenge.
- **Helping**, is for when a player indicate to the others how to do something, such as how to use a controller, what is the correct path to solve the puzzle, or saved the other while they were failing.
- **Global Strategies**, this metric is related events where players had different roles, for example while one traverses the level the other fights enemies.
- **Waited for each other**, was associated to when a player had to wait for another, this was normally due to different levels of skill between the players.
- **Got in each others' way**, is a associated to moments where players wanted to, or did different actions that opposed or hindered progress, or for when one players leads the way and the other lags behind.

For their study they had a group of players play cooperative games and recorded their front and back. Then Magy et al analysed the footage, counting all occurrences of these metrics.

These allowed them to better compare games in terms of how cooperative they were, the more positive CPMs the more cooperative a game would be. These metrics can be applied to compare generated levels to non generated ones when user testing, thus giving us a new way to evaluate how cooperative a generated level is.

2.4 Procedural Content Generator for Cooperative Games

There are Generators that create maps and levels that allow for multiple players, however these levels are generated based on a single player perspective. Games like Minecraft, Risk of Rain¹², use procedural generation to create their maps and levels, they then allow multiple players on these maps, however they are created in a way that does not focus on providing challenges that require cooperation.

Minecraft can be made easier with multiple players, each working on their part of a building, gathering resources or exploring new areas. Risk of Rain follows the same patterns and provides challenges that Rocha et al [18] would classify as Shared Goals. However these are goals that are meant to be achievable alone.

¹²Hopoo Games 2013, Risk of Rain, video game, Microsoft Windows, Chucklefish

The area that combines both Procedural Content Generation and Cooperation has not had much research, van Arkel et al [2] used PCG to generate levels for a simple cooperative game. Their game is a 2D puzzle-platform game for two players, the objective is to move from the start to the end of a level. The players can move, jump, stand on top of each other and interact with levers or move objects.

They use the term collaborative instead of cooperative because they followed the same definition as Zagal et al [16], but as mentioned previously we consider both to be interchangeable.

Van Arkel et al [2] defined game design patterns and for that they followed Reuter et al [20] and Hullett and Whitehead [21] approach of having design patterns describe sections of gameplay, this way all a generator had to do would be to combine them and generate gameplay situations.

The patterns they defined are:

- **The Upsy-Daisy:** An obstacle is unreachable by a normal jump and so the players need to time their jumps, with one on top of the other, to reach the object and push it down.
- **Timed Lever-and-Gate:** A lever that when activated opens a gate for a limited amount of time.
- **Common Enemy:** An enemy that has the side facing the player invulnerable to attacks and so one player must distract it so the other can attack it from behind.

Van Arkel et al [2] used Ludoscope [22] an AI assisted mission and level design tool, it uses graphs of tasks to determine which tasks are reachable after completing other tasks and which goals become unreachable, it can also determine deadlocks. With Ludoscope it is possible to define the process of generating a level, by breaking it down into steps that can be executed separately, and then it uses the principles of model driven engineering and generative grammars to generate levels.

Their level generation process focus on a more automatic approach, as opposed to a mixed authorship, and so does not include much human interaction. It is based on a sequence of steps: Path Generation, Define Level Segments, Apply Design Patterns, Final Adjustments.

It uses a sequence of generative grammars the first receives a small 6x3 grid of undefined tiles where in the leftmost column they place at random a 'Start' tile and then they place at random an 'End' tile on the rightmost column.

The Path generation uses two grammars, the first creates a path from left to right, from the start to the end tile, they then move the 'End' tile and using the second grammar create a path from right to left. This path is generated by replacing the undefined tiles with tiles that have certain orientations for example '1H' represents horizontal segment '1V' a vertical one, 'LCD' stands for Left Corner Down meaning that in that segment the player enters from the left side and would have to move through a corner that exits on the right. They have others like UCR stands for Up-CornerRight, DCR for Down-Corner-Right,

The Define Level Segment expands the smaller grid into a new one where each tile is now 20x20 tile 'segment', it is then chosen a template for each segment depending on their orientation, each orientation

has several templates. Each template can then have 'Encounters' added to them, these are the Game Design patterns described before, 'The Upsy-Daisy', 'Timed Lever-and-Gate', 'Common Enemy'.

The Apply Design Patterns then uses these encounters as smaller level segments to contain the challenges involving certain mechanics. An encounter can have another encounter in it.

The Final Adjustments corrects some objects orientation, and removes unnecessary information. This level is then put through a parser that reads the Ludoscope output file and using the tiles, their position and type the corresponding objects are placed in Unity.

This approach managed to generate many different levels for their puzzle platform game, however it does not allow for much input beforehand from the Game Designer.

Given that the levels generated use templates for each sections and for the challenges, they can generate many different levels however they can become repetitive if they have a small number of templates. These templates are where the game designers have more control when it comes to the level generation.

2.5 Geometry Friends

Geometry Friends¹³ is a cooperative puzzle platform game for two players. The game has two different characters, a yellow circle and a green rectangle. Both characters are subjected to gravity and friction but each character is unique. The circle can jump and its method of movement is through rolling, by jumping it can climb obstacles, this is because its jump height is almost half the height of the level as seen in figure 2.5. The rectangle unlike the circle cannot jump and its movement is by dragging itself across the floor, so to compensate not being able to jump it can change its shape by stretching horizontally or vertically while keeping the same area, so if it is horizontally it will loose height but gain width, it can also climb small obstacles but only if it has enough space, to climb it has to extend upwards and then move towards the obstacle and if it has enough momentum it will fall on top of the obstacle, as seen in figure 2.6. This difference is where the core gameplay lies. The circle is bigger and cannot fit in small places while the rectangle by changing its size can fit through smaller paths. The fact that the characters differ from one another makes this game different from one the van Arkel et al [2] used when testing his generator.

The game is then divided into worlds, these worlds are groups of individual levels that have a theme, such as levels that promote cooperation, others that promote individualism. In each level the players must collect all the purple diamonds, each level differs in the position of the platforms, the platforms type, the characters starting area, and the collectibles position. Some platforms are colored yellow or green and as such the characters that are not those colors cannot pass through them. Figure 2.7 is an

¹³Geometry Friends, <http://gaips.inesc-id.pt/geometryfriends/>



Figure 2.5: Circle jump height

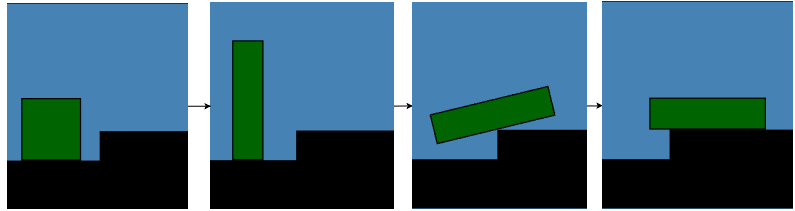


Figure 2.6: Rectangle climbing a ledge

example of a simple level while 2.8 is a more complex level with the colored platforms.

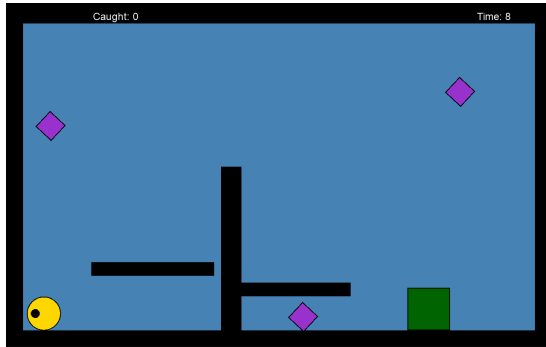


Figure 2.7: Level in Geometry Friends

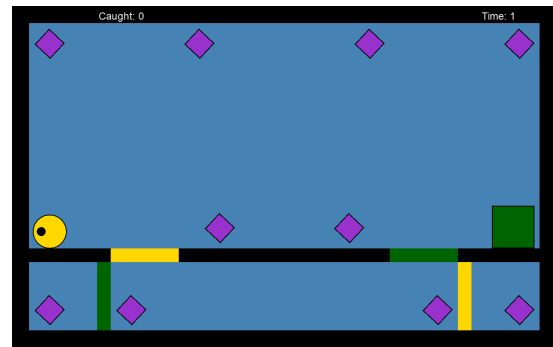


Figure 2.8: Level in with special platforms

The cooperative aspect of the games comes from the interactions needed by both players in order to complete a level, the interactions can be done indirectly, for example a player can reach an area with a collectible that the other cannot, this way by grabbing that collectible the player is helping reach the common goal of finishing the level. Or the interaction can be more directly, and this is the most important interaction as it is the one that better promotes cooperation by having both players do something at the same time. This cooperative action is done by having the rectangle act as a platform for the circle allowing it to reach higher places with its jump, the rectangle can be extra helpful as it can extend upwards as the circle is jumping allowing for an added bonus to the strength of the jump.

2.5.1 A level in Geometry Friends

A level is a limited space with walls surrounding it, any number of platforms, and at least one purple diamond to collect, where any diamond that exists must be reachable. There are also the starting positions, one for the circle and another for the rectangle.

If we were to look at a level in a more abstract sense then we can see it as a set of sequences

of moves that each character can make in order to complete the level, for example the circle can get on top of a specific platform by jumping and then jump again to pick up a diamond, the rectangle can extend horizontally to pass through a small space then move inside to grab a diamond. This way we can describe a cooperative action as a sequence of timed actions, for example, first the rectangle moves underneath a certain diamond, then extends horizontally, to make it easier for the circle to get on top, second the circle gets on top of the rectangle by moving and jumping, third the rectangle extends vertically to its full height, fourth the circle jumps to reach the diamond. For example figure 2.9 shows the solution to solving a level.

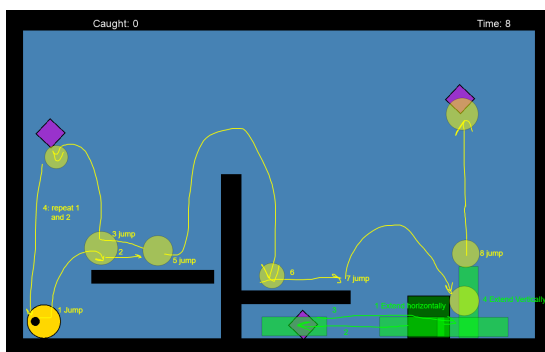


Figure 2.9: Possible Sequence of moves to complete the level

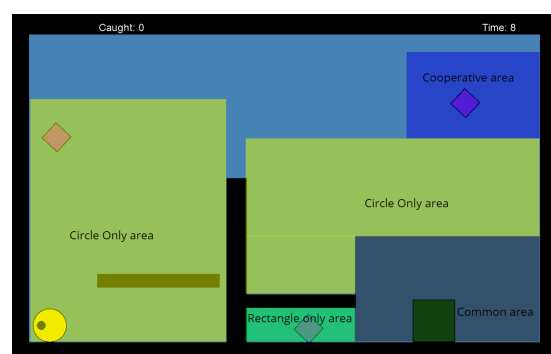


Figure 2.10: Level divided into areas

These actions can be seen as all happening in a contained area related to the diamond we are trying to reach, so we could say that, for each diamond we have an area that is classified on who can reach it, this would lead to four different areas: the circle only area, the rectangle only area, the cooperative area, and the common area. In the first only the circle is capable of reaching that area, the second only the rectangle is capable of it, in the third both are required in order to reach it, and in the last one both can reach it. For example in figure 2.10 we divide the level into those areas.

The levels in Geometry Friends are described by an XML file. This file starts with an element with the tag `<Levels>` encompassing all the levels inside of that world, `<Level1>`, `<Level2>` and so on. Each `<LevelX>` then needs to have at least a set of elements each with the following tags: 'BallStartingPosition', 'SquareStartingPosition' and 'Collectibles', then it can have, even if empty, 'BlackObstacles', 'GreenObstacles', 'YellowObstacles', 'GreenElevators', 'OrangeElevators', and 'TimeLimit'.

The game is played on a 1280 x 800 window, for describing the position of objects it uses the window's pixels as a coordinate system, meaning the top left of the window would be X=0 and Y=0 and bottom right X=1280, Y=800, however since the level is always limited by walls the playable area of each level ends up being 1200 x 720 where each wall is 40 pixels wide the same as the floor and roof, this means that the top left corner of the playable area has X=40 and Y=40 coordinates and bottom right X=1240, Y=760, the width and height also use pixels to represent their size.

As a way to facilitate positioning the characters, if a character has a spawn that puts them inside a platform at the beginning of the level, then that character is pushed away from the center of that platform, so if it close to the top it is pushed upwards, if it close to the bottom it is pushed downwards.

The circle has a 40 pixel radius, so it can be seen as occupying an 80x80 square and its starting position is based on the center of the circle. The rectangle is 100x100 pixels when it starts and it can then change its height to values between 50 and 200, while keeping its area, so when at 50 height it has 200 width and vice versa. Like the circle, the square starting position is based on its center.

The following is the xml corresponding to the level in Figure 2.7.

```
1 <Level1>
2   <BallStartingPosition X="88" Y="712" />
3   <SquareStartingPosition X="984" Y="696" />
4   <Collectibles>
5     <Collectible X="104" Y="280" />
6     <Collectible X="696" Y="728" />
7     <Collectible X="1064" Y="200" />
8   </Collectibles>
9   <BlackObstacles>
10    <Obstacle X="200" Y="600" height="32" width="288" centered="false" />
11    <Obstacle X="504" Y="376" height="384" width="48" centered="false" />
12    <Obstacle X="552" Y="648" height="32" width="256" centered="false" />
13  </BlackObstacles>
14 </Level1>
```


3

Implementation

Contents

3.1	Software	28
3.2	Overview	28
3.3	Genetic Algorithm	29

Van Arkel et al [2]'s approach managed to generate levels but to try and apply it to Geometry Friends, we believe, would not result in the best and diverse levels, this is because the levels do not have as much space nor do they have areas as well defined as his, and our characters are different from each other unlike his where who does what does not matter. We started this project with an idea in mind on how to approach it and, after looking at previous attempts, we decided to use a genetic algorithm to search through the possible solutions to our problem.

3.1 Software

Since Genetic algorithms follow the same procedure, we decided to use already implemented versions. Our first attempt used 'GeneticSharp' a C# genetic algorithm library and our second approach used DEAP [23] a python genetic algorithm library. With 'GeneticSharp' we developed the first iteration of the chromosomes, that used bits, the first implementation of the fitness function and we then used already implemented selection, crossover and mutation methods. We noticed that the fitness attributed to the levels was not always the actual fitness and so after finding this inconsistency we changed implementations. DEAP shared plenty of methods with 'GeneticSharp' but provided an easier alternative to defining the chromosome so we changed from an array of bits to an array of integers.

3.2 Overview

Our goal is to generate levels, specifically for the game geometry friends, so our output will be a level. That includes generating the characters starting position, the platforms, that is their position, width and height, and the collectibles position. To do this we adapted a genetic algorithm where each chromosome would represent everything from the level, so the spawns, platforms and collectibles. We wanted the designer to be able to give some input and be able to guide the algorithm into generating levels with certain characteristics so for this we decided on having the input be a series of areas where the designer would specify how certain areas of the level should be reached. So they would indicate if an area should be reached through cooperation, or if only a certain character should be able to reach that area or both needed to reach it, we would then have the collectibles be evenly spread throughout the different areas. However we later found that just the platforms and spawns positioning was a complex enough problem for our fitness function, so we decided on separating the generating process into two steps, first a genetic algorithm would receive the input and generate the platforms and characters positions that best matched that input, next we would place collectibles in the areas provided by the input. The process is shown in figure 3.1, in (1) is the visual representation of the input, in (2) is the visual representation of the chromosome with the highest fitness at the end of the genetic algorithm, this is what we have at the end

of the first step, a level without collectibles in it. In (3) we have placed the collectibles, they are positioned relative to the input areas and the bigger area has more collectibles, and in (4) we have the level playable inside the game, this meant we had to transform the chromosome and the collectibles position into an xml file that could them be read by the game. We will go into more detail in the following sections.

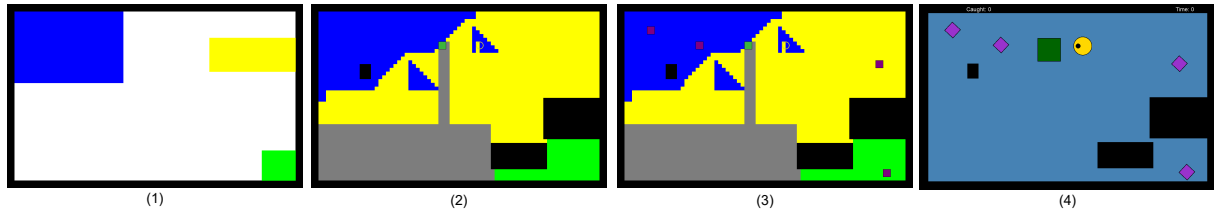
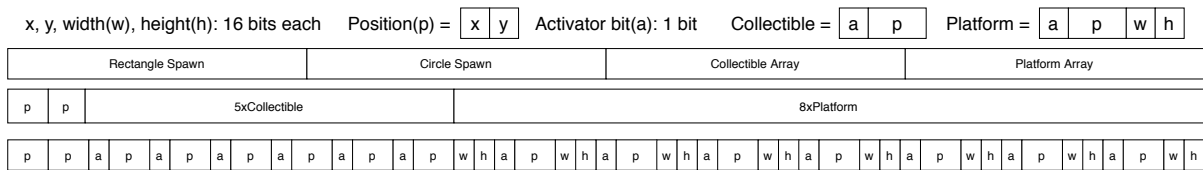


Figure 3.1: The level generation from input to inside the game

3.3 Genetic Algorithm

3.3.1 Chromosomes

In our implementation each chromosome represented a level, the Chromosomes in our first attempt was structured as [Rectangle Spawn, Circle Spawn, Collectible Array, Platform Array]. Both the Rectangle Spawn and Circle Spawn were represented by a position so an 'x' and 'y' value, the collectible array represented the number of collectibles and where they are positioned and the platform array does the same but for the platforms. The collectibles and platforms have one bit, we considered it the active indicator bit, that would determine whether that collectible or platform is actually placed or not, we did this because our chromosome had a fixed size but we didn't want a fixed number of collectible neither a fixed number of platforms. We decided on having a chromosome with a fixed size to limit our search space and also make it easier to manage and alter the chromosomes. The collectibles then also had a position like the spawns, the platforms were like the collectibles but also had the width and height of the platform. The size of the collectible array and the platform array determine the maximum amount of platforms so for the collectible array it could indicate up to 5 collectibles and the platform array could indicate up to 8 platforms. The 'x' and 'y' in the positions each used 2 bytes so 16 bits, this is less than the 32 bits used for an integer value, we choose this because the highest value each 'x' can take was 1280 and for 'y' was 760. The rectangle and the circle spawn each had 32 bits representing them. The collectibles having one bit and a position used $(1+16+16)$ 33 bits, the collectible array used (33×5) 165 bits. The platforms had one bit a position and, for simplicity, the width and height are represented by a position as well, so each platform used $(1+16+16+16+16)$ 65 bits, therefore the total of bits for the platform array was (65×8) 520 bits. The chromosome in total needed $(32+32+165+520)$ 749 bits to fully represent a level.



We later changed the chromosomes because of the fitness function, since the fitness function only looked at the where each character could reach, it did not look at the position of the collectibles. Therefore we decided to separate the level generation into two parts, the first had the level's platforms, rectangle and circle starting position, and in the second part we would add the collectibles. This meant that since the fitness function did not need to take the collectibles into account, neither would the chromosome need to represent them. This was important because when we mutated the chromosome or when we generated children there was a chance it would change only the part related with the collectibles meaning it would, for the fitness function, not make any changes to the level's fitness. The chromosome became [Rectangle Spawn,Circle Spawn,Platform Array], losing 165 bits so it had 584.

When we changed from GeneticSharp to DEAP, we changed the representation from an array of bits to a list of integers, the overall functionality was the same except now we had better control over the values allowing us to have better control over the mutations and crossovers. In both GeneticSharp and DEAP the initial population was generated with random values, all we did was define its size.

3.3.2 Fitness Function

The first fitness function we tested received an array of regions that indicated if only the rectangle should be able to reach it or just the circle or if cooperation was needed or a common region where both should be able to reach, these regions were described by their position, an x and y, their width and height and what type of regions it was. In the figure 3.3 we used an input that requested a cooperative region at the top of the level, shown in blue, a circle only region on the left of the level, in yellow, a rectangle only region on the right, in green, and a common region in grey. The function would evaluate the level using those regions and return a value between 0 and 1, where 1 represented a level that fit the input perfectly.

The first step was to calculate where each character could reach and what places needed cooperation to be reached and only after would it be able to evaluate the level based on the intersection of where each character could reach and the regions indicated. Then the fitness would be the sum of the percentage of the area from all the input regions that matched, shown in the algorithm 3.1. To calculate where each character could reach we used Rafael et al [24] approach, first calculating where each character could fit and then simulating the movements they could make starting from their spawn, this gave us a grid that in each cell we could know who could reach it and how, in figure 3.4 we have a

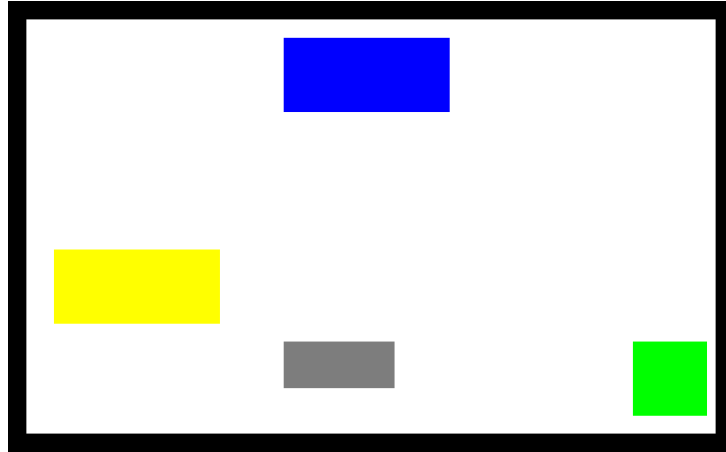


Figure 3.3: Visual representation of input regions for the fitness function

visual representation of that grid and we can see that, in green is where only the rectangle can reach, in yellow is where only the circle can reach, in blue is where the circle can reach with the help of the rectangle, and in grey are areas where both characters can reach. With this grid we could then calculate the intersection with the input regions given by the designer. To calculate the intersection we would go to where the region would be on the level and for each cell inside the region we would compare who could reach that cell with the region type, so if the region type requested cooperation, we would count how many cells could only be reached by the circle with the help of the rectangle and then we would divide that by the area of the input region, giving us the percentage of area intersected.

Algorithm 3.1: Sum of all intersections

Function *FitnessSum(fitnessAreas, level):*

 levelReachabilityGrid = CalculateReachability(level)

 percentageSum = 0

for *area* in *fitnessAreas* **do**

 percentageOfAreaIntersected = CalculateIntersection(levelReachabilityGrid, area)

 percentageSum = percentageSum + percentageOfAreaIntersected

 percentageSum = percentageSum / lenght(*fitnessAreas*)

return percentageSum

In the figure 3.4 we can see the the areas where each character can reach and how, in green only the rectangle can reach those areas, in yellow only the circle, in blue the circle requires the help of the rectangle to reach that area and in grey both can reach. However it is possible to see that near any platform is an area in white that supposedly means no character can reach it, except that in some cases they can, this happens because Rafael's approach first calculates where each character can fit, and if a character would be partially inside a platform it counts that area as somewhere they cannot fit. Then when it goes to simulate where each character can reach it only takes into consideration where they can

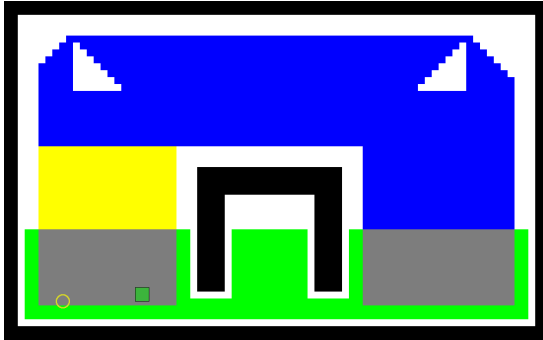


Figure 3.4: Rafaels approach to calculate reachability

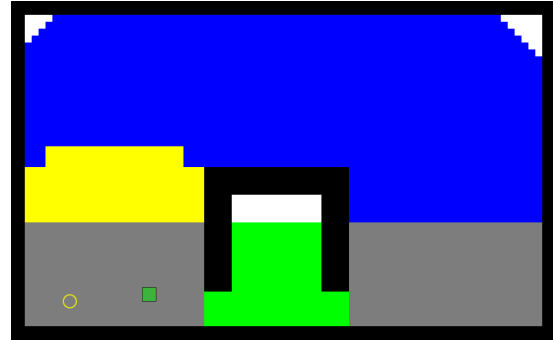


Figure 3.5: Rafaels approach with the improvement

fit, which was previously calculated. This means that near any platform that both could move on top of, would first be a white area and then a rectangle only area and then a common area, in the figure 3.4 we can look at the bottom area of the map and see this happening. When we went to calculate the intersection it would count those areas in green as rectangle only areas but in reality both players can reach those areas. So to fix this problem we added a step that extends their reach. We do this by going to each place that each character can reach and simulate having them placed there, in other words, we go to each position that they can reach and consider that every position around that area they can also reach therefore extending their reach and creating a simulation closer to the game as seen in figure 3.5.

The first implementation of the chromosomes had in consideration where the collectibles could be, however since the fitness function only took into consideration where each character could reach, we did not need the collectibles so we removed them from the chromosome. Another problem we ran into was that to fully simulate where each character could go we would have to take into consideration the level on a pixel scale, which would mean a lot of computation per level, something that Rafael did not do, his approach would take a level and divide it into 16 by 16 pixel wide squares, this meant that a level would go from 1280 pixels of width to 80 blocks and 760 pixels of height to 47 blocks, so when we created the chromosomes and their values could go up to 1280 it meant that in the fitness function it would have to be rounded to a smaller value, so in order to again reduce unnecessary mutations and crossovers we reduced the maximum values for the x and y positions as well as the width and height to fit into the smaller scale.

We tested this approach using the sum of the percentage of intersections and common selection, crossover and mutation methods such as the stochastic selection, two point crossover and the uniform mutation and we saw it would get stuck when generating certain levels. We noticed that it would sometimes ignore one of the areas and maximize the others, for example, if there were two regions indicated, one for rectangle only and another for circle only. It could generate a level where the specified rectangle only region was fully matched but the circle only region was not matched at all, it would give that level a fitness of 0.5, then in subsequent generations it would try to increase the circle only region, but po-

tentially at a cost to the rectangle region which could lead to no overall fitness improvement. This was even more prevalent when one area was smaller than the other, this is because to fully intersect where a character could reach with a smaller area is easier, so that area would more quickly indicate as having one hundred percent coverage while the other area might not have any which would lead to ignoring the larger area in subsequent generations, in order to improve we changed the fitness function. The new approach would still calculate where each could reach and the intersection of the character's reach with the input areas, but instead of adding up the percentage of area matched we would consider the value of the smallest percentage of intersection as show in the algorithm 3.2. This way we would get higher fitness when all of them had reach at least a certain percentage, so while in the previous approach 0.5 could mean one area had zero intersection and another was fully intersected, in this new approach it would mean that every input area had at least fifty percent intersection. This is better since we want the levels to better represent the input.

Algorithm 3.2: Minimum of all intersection

```

Function FitnessMin(fitnessAreas, level):
    levelReachabilityGrid = CalculateReachability(level)
    percentages = []
    for area in fitnessAreas do
        percentageOfAreaIntersected = CalculateIntersection(levelReachabilityGrid, area)
        percentages = percentages + [percentageOfAreaIntersected]
    return Min(percentages)

```

This approach then generated much better results however it created some problems as it took only the worst intersection into consideration, this meant that the others could keep improving but that would not be taken into consideration in the fitness, so we tested with, instead of considering the smallest percentage of intersection, we would instead multiply all the percentages, shown in algorithm 3.3, and that would be the fitness. This approach would give fitness one if every area was fully intersected, fitness zero if an area was being ignored, however while in the minimum intersection approach 0.5 meant that fifty percent of all regions where intersected, in this approach we did not have that guarantee and if every input area had fifty percent the fitness would be 0.5^n where n is the number of input areas which depending on how many areas we specify could be a very low number and might not be a good indicative of how good the level is. This approach took into consideration every time there were improvements on the intersection percentage of any input area and should help create smaller increments in fitness, thus approaching a higher fitness level in a smoother fashion that would be less dependant on mutations making big changes to the levels. It still had some resemblance to the sum of the fitness as, in this approach it could not completely ignore an area, but it would still get stuck because to increase the percentage of an area it could lead to decreasing the percentage in a different area and so the overall fitness might not increase. This did not happen as much with the minimum intersection because we

only look at the smallest value which gave room for the percentage of the other areas to decrease, so long as the smallest percentage increased. Therefore, we believe the guarantee that the minimum intersection approach provides, on how each region is evolving together, is better than having one that had a better intersection and another that had almost no intersection, and so we decided that, in the end, the generator would use the minimum intersection approach.

Algorithm 3.3: Multiplication of all intersections

```

Function FitnessMult(fitnessAreas, level):
    levelReachabilityGrid = CalculateReachability(level)
    percentageMult = 1
    for area in fitnessAreas do
        percentageOfAreaIntersected = CalculateIntersection(levelReachabilityGrid, area)
        percentageMult = percentageMult * percentageOfAreaIntersected
    return percentageMult

```

This fitness function where the input was a set of areas could generate levels and they were varied but depending on the input it could take a very long time to reach acceptable levels. As seen in tables 3.1 and 3.2 with some inputs we could generate levels that met our regions and the results for the same input were quite varied. To create these tables we used the algorithm with the minimum intersection fitness function, an elitism selection method, the uniform crossover we implemented that was specific to our chromosome, and the uniform mutation that either changed the number of platforms or the appearance of the platforms. For example, in table 3.1, for the second input area, we generated different levels, all with the same properties, an area for the rectangle in the middle and a cooperative area at the top, as requested by the input. Looking at another input, the third input, we again generated different levels, but we can see how it can take more generations than others to reach those results, as shown after 100 generations it was not able to create levels with rectangle only areas in the specified regions, while after more generations we get progressively better results. With 500 generations it generated levels where if it blocked the entire bottom third of the level to the rectangle, the input area was achieved, but then with 2000 generations it managed to block only the corners for the rectangle leaving the middle for both. 2000 generations to reach the best results might not be a problem if each generation is fairly quick, but in our case we were testing in a computer with windows 10 and python 3.8, the CPU was an AMD fx 8320, we had 16GB of ram and the project was kept on a 250GB SSD, with a population size of 50 it could take anywhere from 5 seconds up to 9 seconds per generation that is, even in the best case, over 2 and a half hours to generate a level.

The first fitness function allows the designer to give very specific input but it can take quite a long time to generate a level that matches those specifications, so we decided to try a less restrictive fitness function. This was mostly to see what type of levels could be generated if we didn't specify where the areas had to be, and instead we specified what percentage of the level should be those areas. So this

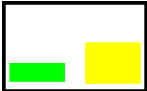
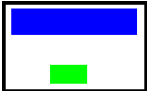

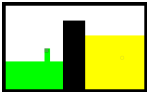
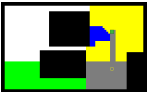

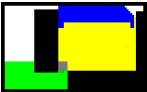



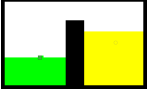



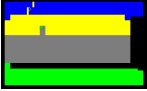
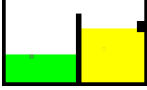
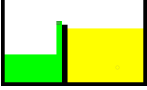




Population Size	50	10	50	10	50	10
Input Area						
100 Max Generations						
500 Max Generations						
2000 Max Generations						

Table 3.1: Example of levels Generated

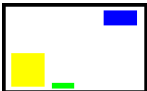

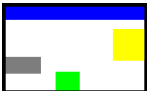








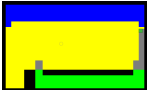
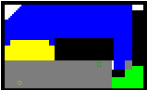







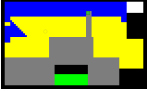
Population Size	50	10	50	10	50	10
Input Area						
100 Max Generations						
500 Max Generations						
2000 Max Generations						

Table 3.2: Example of levels Generated

new function received, as input, a series of percentages, one for each type of area, so a percentage that represented how much area should be for just the rectangle, another for just the circle, another that required cooperation and another for the common areas. We tested two versions of this fitness function, one would consider the percentages relative to the level, so if we specified 30 percent should be rectangle area, then 30 percent of the level would need to be reachable by only the rectangle. The other version was to consider the percentage relative to the area that is reachable. This is less restrictive than the first and easier to specify and imagine since it is not necessary to think of how much 30 percent of a level is and there is no need to take into consideration how much of the level will be occupied by the platforms, but instead the designer can think relative to where the players will be and think of how that area should be used and what for. So when it is specified that 30 percent should be for the rectangle only then that means that of all the areas that can be reached thirty percent of that would be reachable only by the rectangle. We calculate the fitness by subtracting from 1 the absolute of the difference from the expected percentage and the current percentage as shown in 3.1.

$$fitness = 1 - |expRecPer - curRecPer| - |expCircPer - curCircPer| - |expCoopPer - curCoopPer| - |expCommonPer - curCommonPer| \quad (3.1)$$

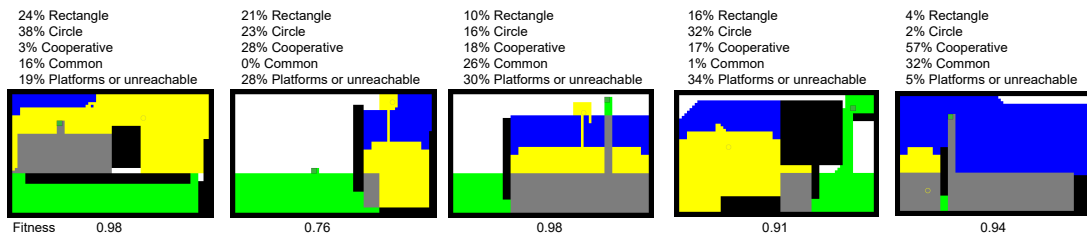


Figure 3.6: Levels generated using the percentage relative to the entire level

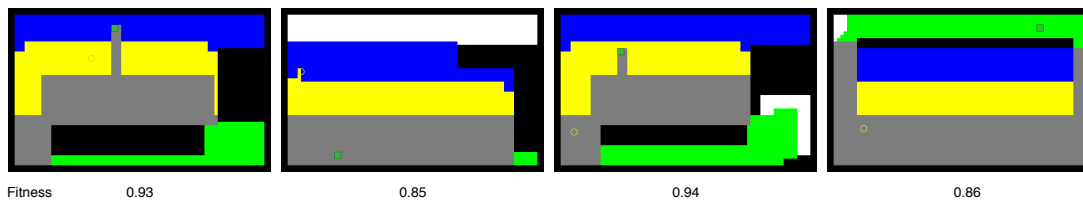


Figure 3.7: Levels generated using 15% rectangle, 20% circle, 20% cooperative, 30% common as the input percentage relative to the entire level

In the figure 3.6 we can see levels generated using the percentage relative to the level approach, the values for these percentage where randomly chosen, and we represent the levels the same as before, so in blue indicates an area that requires cooperation to achieve, in yellow an area that only the circle can reach, in green an area that only the rectangle can reach and in grey an area where both can reach,

in black are the platforms and the rectangle and circle spawns are indicated by a green rectangle and a yellow circle. We used an elitism selection method, a crossover specific to our problem that uniformly decides from which parent to inherit, and a mutation that is also specific to our problem where it either changes the platforms positions or the amount of platforms. From the fitness value we could see that the levels generated match the percentages requested. An interesting thing to note is that, in the second level, it requests zero common areas but you cannot have a cooperative area without a common area, so when it calculates the fitness, it tries to minimize the common area and maximize the cooperative area. Testing with random values showed that it can adapt to any input so we then tested with the same input to see if there was diversity to the levels or if it always found the same solution. In figure 3.7 we show levels that used as input percentage, 15 rectangle only, 20 circle only, 20 cooperative and 30 common, leaving 15 percent for platforms or unreachable places, underneath them is the fitness that they had, these levels where generated after 100 generations. We can see that both levels that have above 0.9 fitness are quite similar. The second level appears to be headed the same direction as the previous two, yet the last level has a much different approach to matching the input. We ran it several times and, for this input, it seemed like the trend for levels with fitness over 0.9 was to have the area for the rectangle at the bottom and then going to the right, as shown in these examples.

When using the second version of this fitness function, where the input is percentage relative to the reachable area and not the entire level, we have a correlation where in order to increase the percentage of a certain type it will always decrease all the others. This happens because the only way to increase percentages are by, either reaching new places or taking from currently reachable places and switching their type of area. When reaching new places, the overall reachable area is larger, so the percentage is relative to that change. This does not happen with the previous approach since the area of the level is of fixed size.

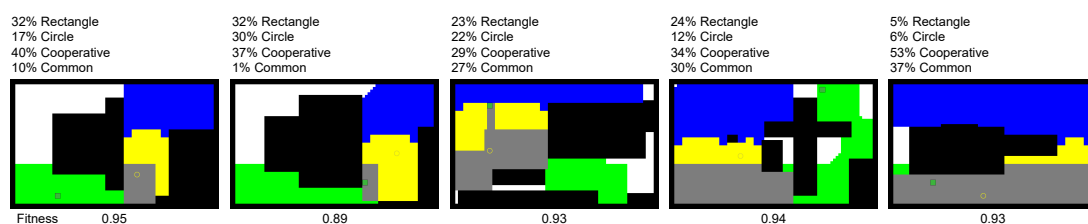


Figure 3.8: Levels generated using the percentage relative to the reachable area

In figure 3.8, we see levels generated using percentage relative to the reachable area, again these values where randomly chosen and through their fitness we could see that they match the input closely, as they all had above 0.85. In the first two examples the levels are quite similar, they are also similar to levels generated using the previous approach. One of their common factors is the input requests a fairly large percentage of rectangle only area and a low percentage for common only area. This can mean that although it is less restricted in its search, in the end the levels generated are not as diverse as we wanted

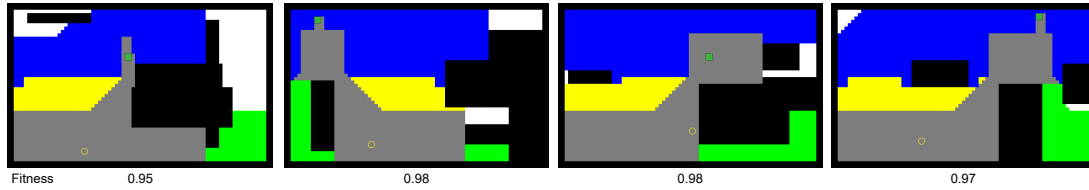


Figure 3.9: Levels generated using 10% rectangle 10% circle 40% cooperative and 40% common as the input percentage relative to the reachable areas

and expected them to be. We then tested with the same input, for this, we used as input percentages: 10% rectangle only, 10% circle only, 40% cooperative and 40% common, these percentages should add to 100 since they do not have to account for the platforms. In figure 3.9, we show the generated levels after 100 generations, underneath each level is their respective fitness, they all have a fitness above 0.9 and we can notice some patterns: they all have the rectangle spawning high, most likely as a way to remove part of the reach from the circle and replace it with a common area, almost all have only one area cut off for the rectangle and this area is to the right, and the platform positioning, appears more diverse than the first fitness function where we specified the areas (in that approach we could understand why the platforms were placed like that).

The trade offs between both approaches are that the first allows for much more control at the cost of needing more time to generate, the second approach while it has more freedom it is too unreliable to produce levels with input from the designer, so it might be better used with random inputs. The first approach if it received random inputs it could have areas overlapping each other, or sticking out of the level, so it is better for helping the designer. That is why we believe that overall the first approach is better for the designer and it is the one we used for the final generator.

3.3.3 Selection

The selection method for choosing the parents originally was to simply take the entire population and use them as parents, randomly choosing two parents and then using the crossover method to create two new members of the population without repeating the parents. This approach was just to have a baseline for comparison, because we knew others should be better. When we look at the graph in figure 3.11, we can see that we have a lot of outliers, this mostly means that the mutation is what is creating those improvements, but since we are not necessarily using them for creating the next offspring they disappear.

When testing all these different selection methods, the random selection method, the stochastic selection method, the tournament selection method with size 4 and size 16, and the elitism selection method, we used a population size of 50, 500 generations, we used the same crossover and mutation methods, as well as the fitness function that received the same areas as input and considered the

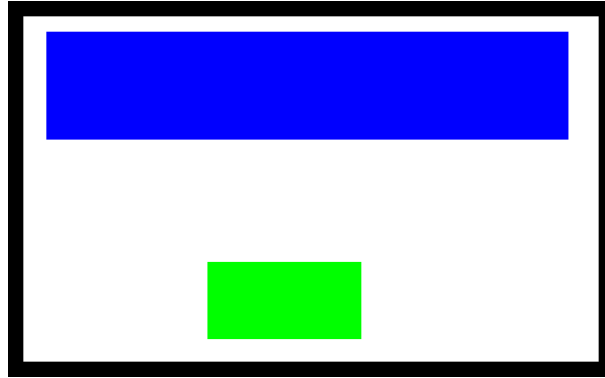


Figure 3.10: Visual representation of input regions used in selection method testing

minimum intersection as fitness, the input areas were one cooperative area along the top of the level and one rectangle only area on the bottom near the middle of the level as seen in figure 3.10. The graphs generated have the generation number along the x axis and the fitness along the y, in blue is the highest level fitness found in that generation, in yellow is the average of all levels fitness values, green is the average of the first quartile, red the second, purple the third and brown the fourth quartile.

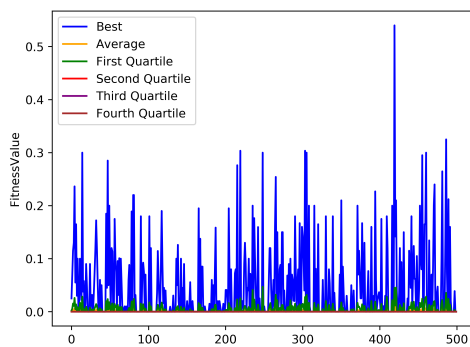


Figure 3.11: Random selection

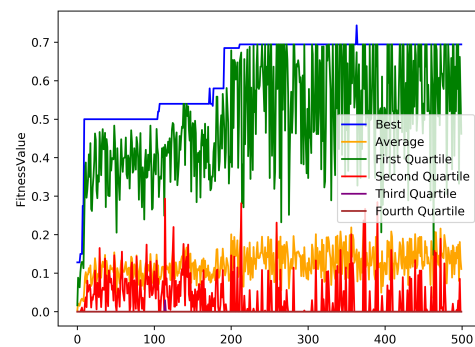


Figure 3.12: Stochastic selection

In order to take fitness into consideration we tried stochastic universal sampling but we ran into a problem where sometimes the whole population had zero fitness. This is because, for example, if every spawn was inside a platform their fitness would be zero, this led to it just choosing random levels and having the same problem as the first attempt. However if one happened to have higher fitness and no other had a fitness value above zero then it would mainly choose that one as both parents and therefore creating only clones of that one as an offspring. We can see in figure 3.12 that, when the fitness is very low, we have the same results as the random approach but, when it is higher, the first quartile is extremely close to the best fitness. This is because the top levels were just copies of the best level because they could have been the result of crossover where the best level was chosen as both parents,

or the top levels were versions of the best level that had different fitness as a result of a mutation. When we implement a function that removes all levels that have the same platforms and spawns and replace them with a random level, we end up with results such as the ones seen in figure 3.13, where we again see results similar to the random selection which indicates that we are relying on the random levels that were added to maintain population size.

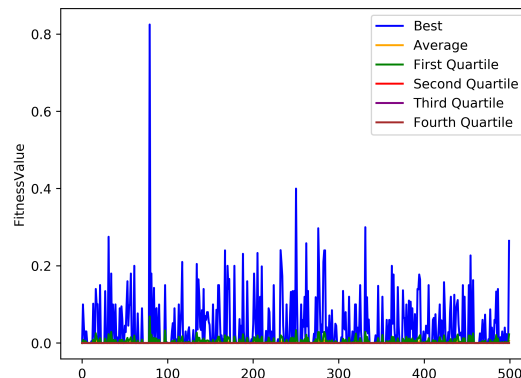


Figure 3.13: Stochastic selection without duplicate levels

We then tried the tournament selection using two different sizes for the tournament, 4 and 16. We found that with the small tournament size we got a similar results as with random selection, although a bit better when looking at the average and first quartile in figure 3.14. With the larger tournament size we ended up having the same problem as with the stochastic selection in which we would again be choosing the same level to be parent several times, and that is not bad as long as it does not create offspring with itself, which happens when using tournament selection for each parent and a larger size. We can see in figure 3.15 that the top levels all have similar fitness, and even though that does not necessarily mean that the levels are the same, when we then look at the entire last generation we saw that the top 12 levels were exactly the same, this also happened with the stochastic selection.

We then tried using elitism, where we would take the top thirty percent of the population and create an offspring while making sure both parents were different and each parent combination would not repeat itself. This, in general, elevated the average fitness as seen in figure 3.16, but after applying mutations it could create worse levels than the previous generations, so we changed it to guarantee that the best level would always stay from one generation to next and it would be used to create offspring. This would mean that that level would not suffer mutations but its offspring would. From figure 3.17 we can see that this results in overall higher fitness values.

Looking at all these graphs, we notice that the third and fourth quartile are both near each other with zero fitness. We believe that this is due to how the fitness function can quickly evaluate a levels fitness as zero. For example, if both characters are inside platforms, then that is an unplayable level

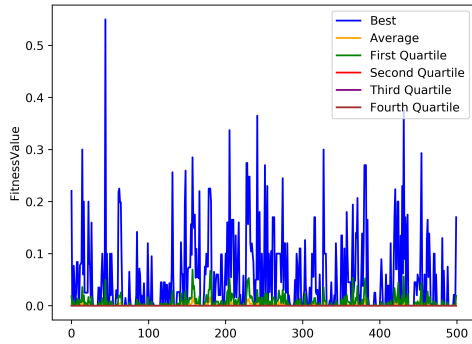


Figure 3.14: Tournament with 4 participants

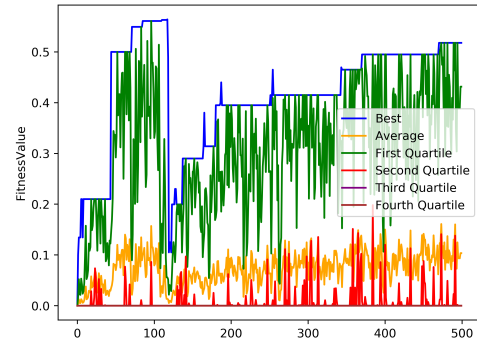


Figure 3.15: Tournament with 16 participants

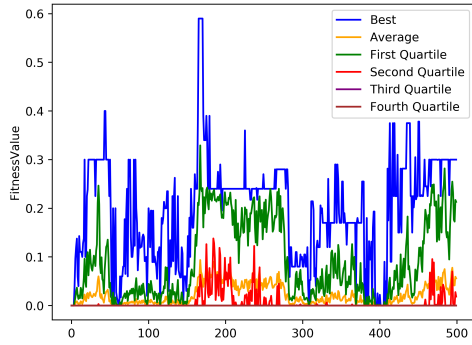


Figure 3.16: Top 30% elitism

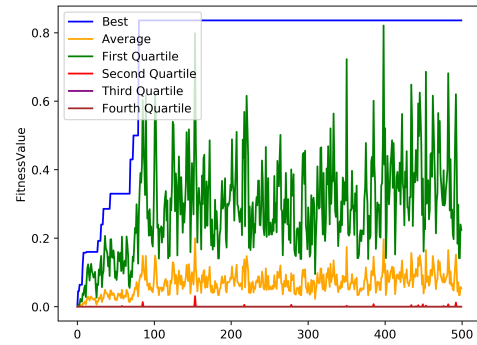


Figure 3.17: Top 30% elitism and maintaining the best

and has zero fitness even if those platforms were positioned in such a way that would otherwise create a high fitness level. As mentioned before, if a character is placed inside a platform, it is pushed either up or down. Unfortunately our fitness function does not take this into consideration and instead just gives it zero fitness. Another thing that gives levels very low fitness is the fact that we use the minimum intersection of the calculated reachability with specified areas, so even if we have a specified area that is fully reachable, but the other is unreachable, be it by having a platform on top of it or simply that characters just can not reach it, then that level also has zero fitness.

3.3.4 Crossover

In the following sections, Crossover and Mutation, will be presented using two levels to help explain what our process was and show how the levels are created, these levels are show in figures 3.18 and 3.19. We will mention how they then create a new offspring inside the algorithm so, in figure 3.20, we show how they are represented as an array of integers(in bold are the activator bits or in this case

A diagram of a simple environment. It features a yellow circle on the left side, a green square on the right side, and a black cross-shaped obstacle in the center. The background is a solid light blue color, and the floor is a solid light gray color.

<i>Level 1</i>	: [20, 42, 7, 42, 1, 28, 25, 4, 18, 1 , 45, 25, 4, 18, 1 , 33, 25, 15, 4, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0]
<i>Level 2</i>	: [61, 43, 5, 40, 1, 12, 37, 18, 2, 1 , 31, 23, 3, 24, 1 , 34, 40, 16, 2, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0]
<i>Level 1 V2:</i>	[20, 42, 7, 42, 1 , 33, 25, 15, 4, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 1 , 28, 25, 4, 18, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 1 , 45, 25, 4, 18]
<i>Level 2 V2:</i>	[61, 43, 5, 40, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 1, 12, 37, 18, 2, 1 , 31, 23, 3, 24, 0 , 0, 0, 0, 0, 1 , 34, 40, 16, 2, 0 , 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0 , 0, 0, 0]

42

The figure consists of two square panels side-by-side, each with a black border. The background of both panels is a solid blue color. In the left panel, a small yellow circle with a black dot in the center is positioned on the bottom-left edge. To its right is a small green square. Further right is a black L-shaped obstacle. In the right panel, the yellow circle has moved to the right, now positioned to the left of the black L-shaped obstacle. The green square has moved to the right, now positioned to the right of the black L-shaped obstacle. This illustrates a sequence of states in a navigation task.

Figure 3.21: Children generated using a one point crossover with Levels 1 and 2 as parents

Figure 3.22: Children generated using a two point crossover with Levels 1 and 2 V2 as parents

second level that were transferred to the children are underlined and, the visual representation for the children is shown.

From the examples shown, we can see that these could be considered mutated levels, since in the first example the third platform from each level switched their x position. The same can be seen in the second example, where the height of the platform was changed and the active indicator was also flipped which resulted in adding a platform that would be completely random, but since these examples have the random numbers as zero the new platform is not shown. Figure 3.22 also shows us how it is relevant that the platforms are not all in the first three platform indexes, if they were, the levels resulting from the crossover would always have three platforms, but, since the platforms'index is not aligned with each other, there are platforms that can be removed or added instead of just switched.

We also tested a more specific crossover to our chromosome. This crossover would switch only platforms that were active or it would switch between platforms that were active in one but not the other, this way it would ignore crossing platforms that were not active and that would not create offspring that did not differ from the parent. It would also, when crossing, take the entire platform so as to not change its size and position.

These approaches would generate two children from the same two parents and the children would be, in a sense, the opposite of the other, because what one child got from one parent, the other would

Level 2 :[61, 43, 5, 40, 1, 12, 37, 18, 2, 1, 31, 23, 3, 24, 1, 34, 40, 16, 2, 0]
 Level 1 V2 :[20, 42, 7, 42, 1, 33, 25, 15, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 28, 25, 4, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 45, 25, 4, 18]
 Cross 1 :[20, 42, 7, 42, 1, 33, 25, 15, 4, 0, 0, 0, 0, 0, 1, 34, 40, 16, 2, 1, 28, 25, 4, 18, 0]
 Cross 2 :[61, 43, 7, 42, 1, 12, 37, 18, 2, 1, 31, 23, 3, 24, 0, 0, 0, 0, 0, 1, 28, 25, 4, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 45, 25, 4, 18]

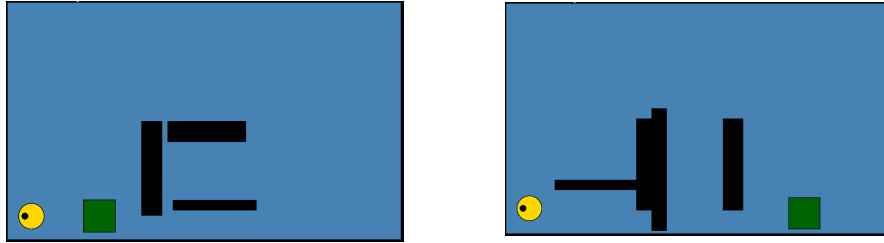


Figure 3.24: Two Children generated with Levels 2 and 1 v2 as parents using the algorithm 3.4

Level 2 :[61, 43, 5, 40, 1, 12, 37, 18, 2, 1, 31, 23, 3, 24, 1, 34, 40, 16, 2, 0]
 Level 1 V2 :[20, 42, 7, 42, 1, 33, 25, 15, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 28, 25, 4, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 45, 25, 4, 18]
 Child 1 :[20, 42, 7, 42, 1, 33, 25, 15, 4, 0, 0, 0, 0, 0, 1, 34, 40, 16, 2, 1, 28, 25, 4, 18, 0]
 Child 2 :[61, 43, 7, 42, 1, 12, 37, 18, 2, 1, 31, 23, 3, 24, 0, 0, 0, 0, 0, 1, 28, 25, 4, 18, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 45, 25, 4, 18]

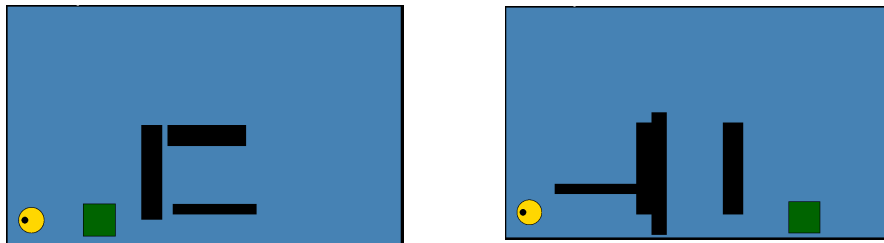


Figure 3.25: Levels 1 and 2 after uniform mutation

3.3.5 Mutation

The mutation that was used at the beginning was a simple bit flip mutation that would randomly choose a bit and flip it. With so many different bits it could make little to no difference in the outcome so we then tried with a uniform mutation that would go through each bit and with a random chance it would flip that bit.

Both these mutations had the same problem and that was because our chromosome had the active indicator bit that would effectively make a lot of other bits irrelevant. Additionally when we still had the collectibles in our chromosome, it could change the collectible part only, this meant that after a mutation the way the chromosome was evaluated could remain the same. When we changed chromosome representations to an array of integers without the collectibles, since we no longer had bits, we tested what we consider something equivalent to the bit flip and that was using a random integer. The first approach was to choose a random index and generate a random integer, the second was to do that uniformly through the array and these still had the same problem. We can see, in figure 3.25, the changes that the uniform mutation can cause, the values underlined are the ones that were altered through the mutation, some activator bits where changed, in this case it is not noticeable because other values, such as width or height, are zero, however, when a chromosome is first created, they have random integers, so new platforms with random values would have been added.

reachable by the area type request and then the algorithm checks if it is not near other collectibles. If one or more of these conditions fail, it generates a new random position inside the area and repeats the process, after a certain amount of failed attempts it does not place the collectible, meaning that area will have one less collectible than what was decided based on the area's size. In figure 3.28, we can see a level generated with the input shown in figure 3.10 and collectibles placed according to the same input areas.

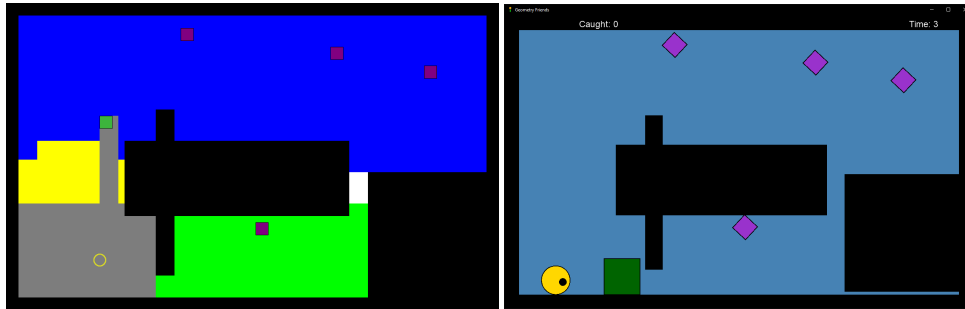


Figure 3.28: A level with collectibles placed and its representation in game

4

Evaluation

Contents

4.1 Metrics	50
4.2 Experimenting	51
4.3 Level Generator	53

After all the iterations, we decided that the end generator would use the fitness function that received, as its input, a series of areas, it would use elitism as a selection method, the crossover would only generate one child and it would uniformly choose from which parent it would inherit an attribute, the mutation, we decided on, would change either changed the number of platforms or uniformly change the aspect of the platforms. This version of the generator was the one used for testing¹.

4.1 Metrics

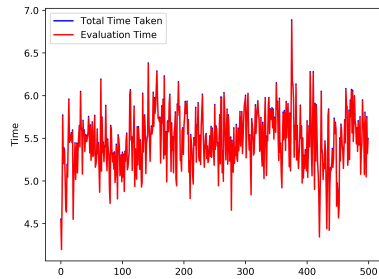


Figure 4.1: Time Taken per generation with population of 50

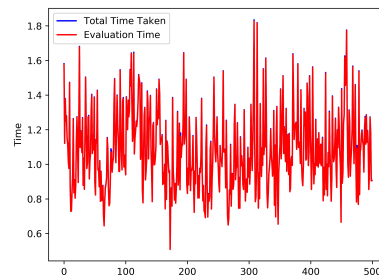


Figure 4.2: Time Taken per generation with population of 10

We did a short study of the generator based on the time it took from receiving the input to having a level generated. We tested in a computer with windows 10 and python 3.8, the CPU was an AMD fx 8320, we had 16GB of ram and the project was kept on a 250GB SSD. We found that with a population size of 50 each generation took on average 5.5 seconds where the majority of that time was used evaluating the fitness of the levels, while with a population size of 10 it took on average 1.1 second per generation. In figures 4.1 and 4.2 we can see that the time per generation varies a lot. This happens because during the evaluation we can determine at earlier points if the level will have a certain fitness, for example, if no spawns are valid then the fitness for that level will be zero, others then take more time because the bigger the amount of a level is reachable the longer it takes to evaluate. The average time to calculate the fitness of a level is one tenth of a second but, in figure 4.3, we can see that the lowest values are about 0.05 seconds, while the highest can go a bit above 0.2 seconds which is at least four times longer than the fastest levels.

¹ <https://github.com/NMBLM/GeometryFriendsLevelGenerator>

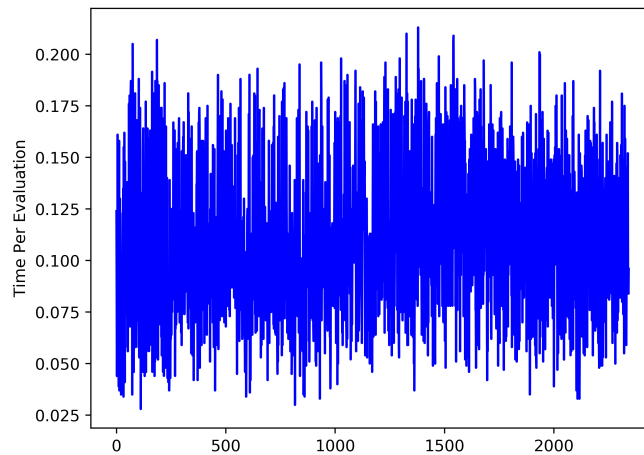


Figure 4.3: Time per evaluation of a level

4.2 Experimenting

We asked people to experiment with the generator, and give us some feedback. The process started by showing them the game and having them play the game. Then we introduced them to generator, we explained how it worked, by showing examples of input and examples of output, and showing how they where related. Next, we asked them to provide input for the algorithm, in order to do this we created a basic GUI tool, shown in figure 4.4, it was developed in python using ‘tkinter’, its main purpose was to provide the participants with a visual representation of what their input meant, so it showed where in the level and what type of area they where requesting. This tool would take a series of inputs describing the regions and then generated those regions in the image on the right, the add row button allowed the tester to specify more areas, the confirm spec button updated the preview image on the right, that allowed them to see where in the level they were specifying the area, the save spec created a file that could then be used as input for the level generator. We then used that input to generate levels. The generator then chose the best level generated and placed collectibles in it. The result was 10 versions of the best level where only the collectible placements changed. We then had them choose the version that seemed the best in terms of collectible placement and we had them play the generated level and say how it matched with their expectation.

The feedback we were looking for was, firstly, if the levels generated meet their requested input, secondly, if the levels were playable, thirdly, if the input requested was meaningful (as in, if it helps define what the designer wants when creating a level), fourthly, what their opinion was on the time it took to generate the levels after giving the input.

The responses varied a bit, depending on the input, the levels generated could be quite good and

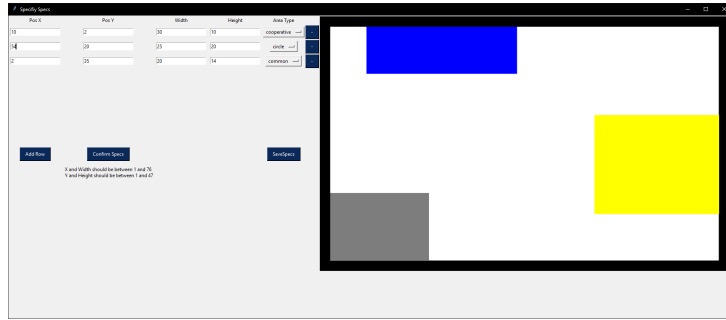


Figure 4.4: GUI Tool to specify input, with some example input

playable or they could be impossible to complete. Some inputs requested impossible combinations such as an area just for the circle directly above an area that requires cooperation to reach, this type of inputs lead to the generator not being able to create a level meeting the requirements.

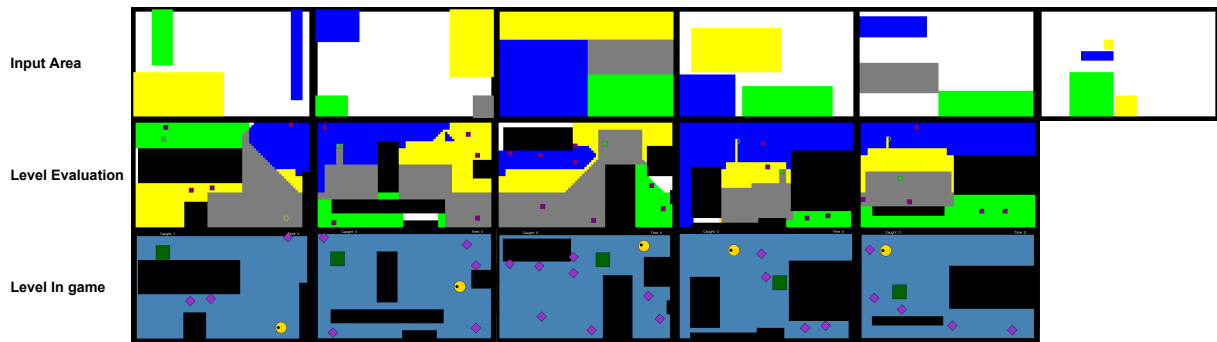


Figure 4.5: Examples of input from testers and the final level generated

We had some positive reactions as the levels generated did somewhat meet what the participants were expecting, but its important to note that those that tested the tool are not familiar with the game and asking them to create a level for a game they played only minutes before can result in unusual inputs. In the figure 4.5 we show some of the inputs given to us in the test and the levels generated at the end, including their in game representation. Our generator for the last input did not manage to generate a level that would satisfy it and so even after the 500 generation all levels had a fitness value of zero, we believe this was due to having a small circle only area above a cooperative area.

As for if the levels were playable most of them could be completed, but for example the third input requested generated a level where the rectangle could either go left or right but not both ways, and so it was not possible to complete.

When asked if the input requested was something helped define their vision for the level, we had mixed responses, some said that it would be better to specify the actions instead, so, for example, saying they wanted the circle and the rectangle to cooperate twice and how they should cooperate. Others said that it was abstract enough and that if they wanted to be more specific, it would be better to

just create the level entirely.

Lastly, when asked about the time it took to generate the level every one agreed that it was too long, even if it is to be done during development and not right before the level was going to be played.

4.3 Level Generator

With the generator complete we can classify it based on previously defined properties. In terms of Speed it is not fast, it cannot generate in real time and therefore the generation has to be done beforehand during the development, this does allow for the designer to then make minor changes on the end level. When we look at Reliability we can say that it is not entirely random, this is because it checks where each character can reach in the levels and equates that to whether it can be completed or not, it does this by calculating where each character can reach and then guaranteeing that the collectibles are placed in reachable areas. When calculating where each character can reach it assumes that the character can go in any direction and then return to where they started. This is not always true as, for example, we can have the rectangle spawn on top of a platform that separates the level in two, an area on the left and an area on the right. When we calculate the reachability it determines that the rectangle can reach both the left and right area, however when we play the level, if we decide to go to the area on the left, we will have to fall and, since the rectangle cannot jump, we will not be able to go back on top of the platform. Therefore without restarting the level we cannot reach both areas, so for the most part it can remove most levels that cannot be finished but not all, we could see this happen in one of our tests with users shown previously. In terms of Controllability, the designer can specify areas of interest, as previously mentioned the areas that determine who should be able to reach them and how, so it does provide some control to the designer. In terms of expressivity and diversity we saw that given the same input it can generate different levels, however since the inputs, in a sense, specify how a level should be completed, it might be considered the same level in a more abstract way, but because we can specify many different inputs we would say that it does provide expressivity and diversity. Lastly in terms of creativity and believability we would say that the positioning of the platforms and of the collectibles do reveal that the levels were generated by a machine, this is because in most human made levels, there is some sense of symmetry or order, such as collectibles that are at a certain height will all be lined up, whereas with our collectible positioning they will be very obviously misaligned.

5

Conclusion

Contents

5.1 The Final Generator	55
5.2 Future Work	55

In this work we proposed a level generator that could provide cooperative challenges in its levels and developed a level generator for the game Geometry Friends. The method for creating those levels used a genetic algorithm with a chromosome that represented the level as the solution, then a fitness function that received abstract input, such as where certain events should take place (in our generator these were the area that specified cooperation or individual tasks), the function then evaluated the levels in terms of where the cooperative events and individual tasks were occurring and how those compared to the requested input. For a problem as complex as this, we believe that the crossover and mutation methods should be tailored to the chromosome and that common methods might not be enough.

5.1 The Final Generator

In the end, we created a generator that could receive input from the designer specifying areas of interest. These areas could represent things such as: only the rectangle should be able to reach this area, or only the circle should be able to reach this area, or both characters should be able to reach this area, or, finally, this area should be reachable only by having both players cooperate. The generator could create levels that matched those inputs. To do this it used a genetic algorithm, the chromosome represented the level and specified its features, it then evaluated where each character could reach in the level, then compared that to the input given by the designer and gave it a fitness value. To get the best results and improve the search done by the genetic algorithm we studied the selection methods and found that elitism provided better results, we found that the crossover method was better if it was specific to our level and we came to the same conclusion regarding the mutation, both needed to take in consideration the features present in our chromosome. In a second step after the genetic algorithm created a level, we placed collectibles in the generated level according to the input areas.

5.2 Future Work

The most direct improvements that can be made are improvements to the current generator. As it was described it cannot take into consideration all types of platforms that are available in the game, so extending it to be able to generate levels using the yellow and the green platforms is one possible improvement, another possible change is to have the chromosomes not be of fixed size and allow it to have more than eight platforms. One thing that could be changed is the input, we tested two different types of inputs, but others can be explored as well, for example, one where the designer defines the solution by indicating which moves to make and then the generator only guarantees that that path is possible and that it is a solution to the level. The levels we generated did not focus on appearing human made, this can be another point of study.

As for the area of cooperative level generators, there is still a lot of work that can be done and different approaches that can be tested. For more complex games, it might not be possible to calculate where each character can be and where, so an approach that used intelligent agents could be developed to play those games and try to complete the levels generated, of course that would require an artificial intelligence that is capable of completing cooperative challenges, these types of AI are very hard, especially ones that require timed actions on the part of both players. Using neural networks to evaluate the levels is a possible way to potentially speed up the evaluation process, the biggest hurdle in this approach would be to have an extensive enough set of levels that are also evaluated.

Bibliography

- [1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*, 1st ed. Springer Publishing Company, Incorporated, 2018.
- [2] B. V. Arkel, D. Karavolos, and A. Bouwer, "Procedural generation of collaborative puzzle-platform game levels," 2015.
- [3] M. Hendrikx, S. Meijer, J. Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications and Applications (ACM TOMCCAP)*, vol. 9, no. 1, pp. 1:1–1:22, Feb. 2013.
- [4] B. Brathwaite and I. Schreiber, *Challenges for Game Designers*, 1st ed. USA: Charles River Media, Inc., 2008.
- [5] E. Falkenauer, "A hybrid grouping genetic algorithm for bin packing," *Journal of Heuristics*, vol. 2, pp. 5–30, 1996.
- [6] "A System for Automatic Construction of Exam Timetable Using Genetic Algorithms," *Tékhné - Revista de Estudos Politécnicos*.
- [7] A. Balali Moghadam and M. Kuchaki Rafsanjani, "A genetic approach in procedural content generation for platformer games level creation," 02 2017.
- [8] A. Connor, T. Greig, and J. Kruse, "Evolutionary generation of game levels," *EAI Endorsed Transactions on Serious Games*, vol. 5, p. 155857, 04 2018.
- [9] F. Mourato, M. Santos, and F. Birra, "Automatic level generation for platform videogames using genetic algorithms," 11 2011, p. 8.
- [10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
- [11] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

- [12] Y. Lavinas, C. Aranha, T. Sakurai, and M. Ladeira, "Experimental analysis of the tournament size on genetic algorithms," in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2018, pp. 3647–3653.
- [13] I. M. Oliver, D. J. Smith, and J. R. C. Holland, "A study of permutation crossover operators on the traveling salesman problem," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc., 1987, p. 224–230.
- [14] D. E. Goldberg and R. Lingle, "Alleles and the traveling salesman problem," in *Proceedings of the 1st International Conference on Genetic Algorithms*. USA: L. Erlbaum Associates Inc., 1985, p. 154–159.
- [15] V. Cicirello, "Non-wrapping order crossover: An order preserving crossover operator that respects absolute position," vol. 2, 07 2006, pp. 1125–1132.
- [16] J. Zagal and J. Rick, "Collaborative games: Lessons learned from board games," *Simulation & Gaming - Simulat Gaming*, vol. 37, pp. 24–40, 03 2006.
- [17] J. Marschak and R. Radner, "Economic theory of teams," no. 59a, 1972, Cowles Foundation Discussion Papers. [Online]. Available: <https://ideas.repec.org/p/cwl/cwldpp/59a.html>
- [18] J. B. Rocha, S. Mascarenhas, and R. Prada, "Game mechanics for cooperative games," 2008.
- [19] M. Seif El-Nasr, B. Aghabeigi, D. Milam, M. Erfani, B. Lameman, H. Maygoli, and S. Mah, "Understanding and evaluating cooperative games," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 253–262. [Online]. Available: <https://doi.org/10.1145/1753326.1753363>
- [20] C. Reuter, V. Wendel, S. Göbel, and R. Steinmetz, "Game design patterns for collaborative player interactions," in *DiGRA*, 2014.
- [21] K. Hullett and J. Whitehead, "Design patterns in fps levels," in *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, ser. FDG '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 78–85. [Online]. Available: <https://doi.org/10.1145/1822348.1822359>
- [22] J. Dormans, "Engineering emergence: applied theory for game design." Universiteit van Amsterdam [Host], 2012.

- [23] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, “DEAP: Evolutionary algorithms made easy,” *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.
- [24] R. V. P. de Passos Ramos, “Procedural content generation for cooperative games,” Master’s Thesis, Instituto Superior Técnico, Universidade de Lisboa, Nov. 2015.

