Procedural Content Generation for Cooperative Games

Nuno Martins
nuno.lages.martins@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2020

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 in Games, Procedural Content Generator for Cooperative Games, Genetic Algorithm, Geometry Friends

1. Introduction

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. 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 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.

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.

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

2. 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

3. Related Work

3.1. 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 [13] many games such as Civilization[1] and No Mans Sky[2] uses it to

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

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

create their worlds. Togelius et al [13] defined content as most of what is in a game, except things like the game engine and Non-playable Character Artificial Intelligence behaviour.

When developing new tools it is important to be able to classify them, so some desirable properties for a PCG have been defined [13] as: Speed, Reliability, Controllability, Expressivity and diversity, and Creativity and believability. Togelius et al [13] then used those properties and defined a taxonomy of PCG that consists of the following dimensions: Online vs Offline, Necessary vs Optional, Degree and Dimension of control, Generic vs Adaptive, Stochastic vs Deterministic, Constructive vs Generate-and-test, Automatic generation versus mixed authorship.

3.2. Content for Procedural Generation

The definition of content by Togelius et al [13] is too broad, Hendrikx et al [6] 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. These layers tend to have concrete and abstract elements, concrete ate things or objects that can be interacted with while abstract are more visual and concept driven.

There are many different ways of creating a Procedural Content Generator, Hendrikx et al [6] 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, Generative Grammars, Image Filtering, Spatial Algorithms, Modelling and Simulation of Complex Systems, Artificial Intelligence.

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 used in PCG. Connor et al [2] 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. Mourato et al [9] applied a genetic algorithm to generating levels for the 1989 Prince of Persia[3] a 2d platforming game. Given these varied uses of the genetic algorithms we believed it could generate good results for our problem as well.

[3]Jordan Mechner, Brøderbund Software,1989, Prince of Persia, MS-DOS, Brøderbund Software, Ubisoft

3.3. Genetic Algorithms

Genetic algorithms fall into the Artificial Intelligence group of algorithms, according to Hendrikx et al[6]. This is because the idea behind them is to mimic biological evolution. Goldberg et al[5] defines them 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 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.

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.

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.

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. Exam-

ples of these selections methods are the Roulette Wheel, Rank, and Stochastic Universal Sampling selection. Others compare fitness value directly such as the Tournament Selection and Elitism selection method.

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, there is also a K-point crossover variant. Another crossover method is the uniform crossover, this method creates offspring where each gene is chosen with a certain probability, normally an equal probability, which parent it inherits from. Something that can be necessary is 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.

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. Other variations of the flip bit 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. Other methods involve switching the order, some reverse the gene order, others reverse a segment, some shuffle the index of the genes.

3.4. 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. Zagal et al [14] 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 [8] 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. 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.

Cooperation in video games comes in many different ways and Rocha et al [11] defined several common Design Patterns and Challenges Archetypes that appear in video games. These design patterns are: Complementarity, Synergies between abilities, Abilities that can only be used on another player, Shared Goals, Synergies between goals, Special Rules for Players of the same Team. These where later extended by Magy et al [12] where they added: Camera Setting, Interacting with the same object, Shared Puzzles, Shared Characters, Special characters targeting lone wolf, Vocalization, Limited Resources.

Reuter et al [10] and Hullettand and Whitehead [7] 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 Rule.

3.5. 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 Rain[4], 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.

The area that combines both Procedural Content Generation and Cooperation has not had much research, van Arkel et al [1] 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. Van Arkal et al [1] defined game design patterns and for that they followed Reuter et al [10] and Hullettand and Whitehead [7] approach of having design pat-

[4]Hopoo Games 2013, Risk of Rain, video game, Microsoft Windows, Chucklefish

terns describe sections of gameplay, this way all a generator had to do would be to combine them and generate gameplay situations. Van Arkel et al [1] used Ludoscope [4] an AI assisted mission and level design tool.

3.6. Geometry Friends

Geometry Friends[5] 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 the rectangle cannot jump, but 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. 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. In each level the players must collect all the purple diamonds, there are different platforms types, some platforms are colored yellow or green and as such the characters that are not those colors cannot pass through them. Figures 1 and 2 are examples of levels.

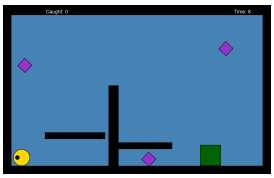

Figure 1: Level in Geometry Friends

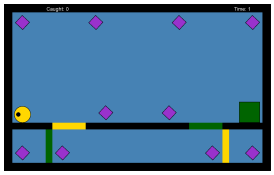

Figure 2: Level in with special platforms

4. Implementation

Van Arkel et al[1]'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 on how to approach in mind and after looking at previous attempts we decided to use a genetic algorithm to search through the possible solutions to our problem.

4.1. 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, 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.

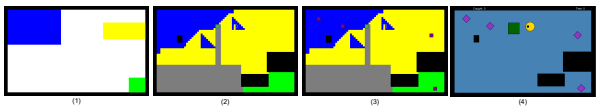

Figure 3: The level generation from input to inside the game

4.2. 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

[5] Geometry Friends,http://gaips.inesc-id.pt/geometryfriends

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. We later changed the chromosomes because

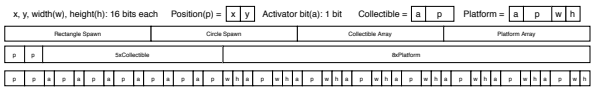

Figure 4: First Chromosome genotype

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 the chromosome did not need to represent them.

4.3. 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 5 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.

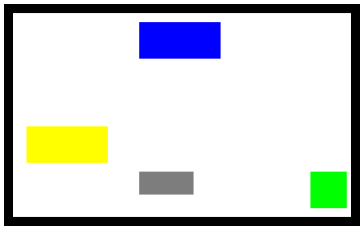

Figure 5: Visual representation of input regions for the fitness function

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. To calculate where each character could reach we used Rafael et al [3] 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 6 we have a 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.

In the figure 6 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 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 figure6 we can look at the bottom area of the map an 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

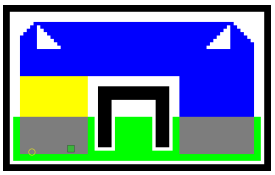

Figure 6: Rafaels approach to calculate reachability

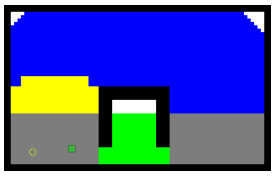

Figure 7: Rafaels approach with the improvement

area they can also reach therefore extending their reach and creating a simulation closer to the game as seen in figure 7.

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, in subsequent generations it would try to increase the circle only region, but potentially at a cost to the rectangle region which could lead to no overall fitness improvement. 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 between the all input regions. This way we would get higher fitness when all of them had reach at least a certain percentage, this was better since we want the levels to better represent the input.

This approach then generated much better results, but 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 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. This approach took into consideration every time there where improvements on the intersection percentage of any input area and should help create smaller increments in fitness leading to a smoother approach to a higher fitness level that would be less dependant on mutations that made big changes to the levels, like the previous approach, yet it still had some resem-

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

Table 1: Example of levels Generated

blance 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 that 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.

This fitness function where the input was a set of areas could generate levels and they where varied but depending on the input it could take a very long time to reach acceptable levels. As seen in table 1, with some inputs we could generate levels that met our regions and the results for the same input where quite varied, to create this 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. We can then see that for example in table 1, for the first 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 the other input, the second 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 that way 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.

4.4. Selection

The selection method for choosing the parents originally was just to take the entire population and use them as parents, randomly choose 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 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 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.

To take fitness into consideration we tried stochastic universal sampling but we ran into the problem that 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 fitness then it would mainly choose that one as both parents and therefore creating only clones of that one as an offspring.

We then tried the tournament selection we used 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 we looked at the average fitness. With the larger tournament size we end 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 again not bad as long as it does not create offspring with itself, which with a tournament for each parent and a larger size lead to that

We then tried using elitism, where we would take the top 30% of the population and create an offspring while making sure both parents where different and each parent combination would not repeat itself. This in general elevated the average fitness as seen in figure 8, 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 9 we can see that this way overall the values are higher.

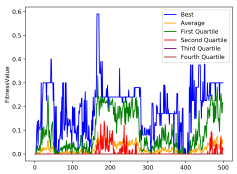

Figure 8: Top 30% elitism

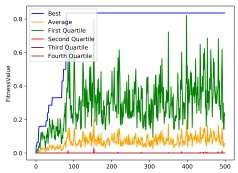

Figure 9: Top 30% elitism and maintaining the best

Looking at 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 cutoff a levels fitness to zero very quickly, for example if both characters are inside platforms, then that is an unplayable level and has zero fitness even if perhaps those platforms ended being positioned in such a way that would otherwise create a high fitness level. 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.

4.5. Crossover

The crossovers we originally tested were the one point crossover and the two point crossover. They would take two parents and create two children. Like in the mutation, but less likely, this could generate children that were equal to the parents, for example if both parents had only the first three platform active and the point chosen was after those three platforms the crossover would just change the platforms that where not active creating offspring that were evaluated the same as the parents. Another problem of choosing a random index was that it could pick an index in the middle of a value (x, y, width, height) from a platform and could change the platform itself acting almost like a mutation,

that is an unwanted side effect. Since we did not want those side effects we tested a more specific crossover to our chromosome, it 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 get from the second and vice versa. So the other crossover we tested was to choose two parents and generate only one child we would for each feature, rectangle spawn, circle spawn, platform, we would give it 50% chance to be from the first parent or the second parent.

By generating only one child it is possible to not repeat parents, while in the previous approaches two parents would generate two children, in this you can repeat one parent and not the other, allowing you to choose a parent you believe will be better to generate offspring and let them cross with a more varied selection, rather than repeating both parents.

4.6. 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 or 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 a random integer, so 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.

Therefore the next mutation we tested was more specific to our chromosome, it could go to the active indicator for the platforms and it would flip it, or it could choose a platform that was active and go through its x, y, width and height and change them, this was the equivalent to the flip bit, but would guarantee that the level would always be changed and therefore change its evaluation. This mutation did not alter the level much which meant that it would take a lot of mutations to get some significant changes. That is why we then implemented a variation on the uniform mutation in such a way that would guarantee changes, it would either change the number of platforms, or it would change the platforms or character starting positions(spawns). We believe this approach is better than uniformly changing all its values because it could lead to too many mutations. Separating it into either changing only the platforms number or only the platforms themselves gives us a bit more control over the evolution.

4.7. Collectibles

The levels generated do not yet have the collectibles placed so in the following step we inserted the collectibles in the level by using the input areas as the regions to place them in, the requirements to placing a collectible were: it must be in an area that can be reached by a character, they should be spread evenly through all the input areas, if an area has a very large area then it should have more collectibles, we set area size thresholds for the amount of collectibles each region had, first every region had at least one collectible, then if a region occupied more 10% of the level it would have 2 collectibles, if it represented more than 20% it would have 3 collectibles, then if it represent more than 35% it would have 4 collectibles and any region had an area that would be equal to 50 or more% of a level would have 5 collectibles. The algorithm places collectibles in each area, to do that, for each area it first checks how many collectibles to place in it, it decides based on the region's size, then for each collectible it randomly chooses a position inside the corresponding area and place them there, whenever a collectible is placed it checks if it is in a reachable position, if the position is reachable by the area type request and then if it is not near other collectibles, if one or more of these conditions fail, it generated 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 then what we decided based on its size. In figure 10 we can see a level generated with the same input used in testing the different selection methods, and collectibles placed according to the same input areas.

5. Results

After all the iterations we the decided that the end generator would use the fitness function that received as input a series of areas, it would use elitism as a selection method, the crossover only gener-

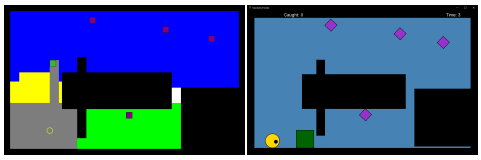

Figure 10: A level with collectibles placed and its representation in game

ated 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 changed the aspect of the platforms. This version of the generator was the one used for testing[6].

6. Metrics

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, with a population size of 10 it took on average 1.1 second per generation. The time per generation can vary 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 the lowest values are about 0.05 seconds while the highest can go a bit above 0.2 seconds that is at least four times longer than the fastest levels, and we want the levels to have reachable areas so taking longer represent levels that have more reachable places.

7. 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 to do this we created a basic GUI tool, 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.

What we where looking as feedback was, first, if the levels generated meet their requested input, second, if the levels were playable, third, if the input requested was meaningful, as in, if it helps define what the designer wants when creating a level, fourth, 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 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.

For what we were looking for first we had some positive reactions as the levels generated did somewhat meet what the participants where expecting, but its important to note that those that tested the tool are not familiar with the game and so asking to create a level for a game they played only minutes before can indicate the unusual inputs. In the figure 11 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 it was due to having a small circle only area above a cooperative area.

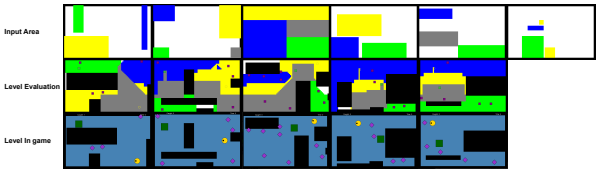

Figure 11: Examples of input from testers and the final level generated

[6]https://github.com/NMBLM/GeometryFriendsLevelGenerator

As for if the levels where 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. We 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 that it would be better to just create the level entirely. Lastly when asked about the time it took the generate the level every one agreed that it was too long, even if it is to done during development and not right before it was going to be played.

8. Conclusions

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 where 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.

9. 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.

10. 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 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.

Acknowledgements

Gostaria de agradecer ao Rui Prada e José Rocha pela a 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.

References

[1] B. V. Arkel, D. Karavolos, and A. Bouwer. Procedural generation of collaborative puzzle-platform game levels. 2015.

[2] A. Connor, T. Greig, and J. Kruse. Evolutionary generation of game levels. *EAI Endorsed Transactions on Serious Games*, 5:155857, 04 2018.

[3] R. V. P. de Passos Ramos. Procedural content generation for cooperative games. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, Nov. 2015.

[4] J. Dormans. Engineering emergence: applied theory for game design. 2012.

[5] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[6] 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)*, 9(1):1:1–1:22, Feb. 2013.

[7] K. Hullett and J. Whitehead. Design patterns in fps levels. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, page 78–85, New York, NY, USA, 2010. Association for Computing Machinery.

[8] J. Marschak and R. Radner. Economic theory of teams. 1972.

[9] F. Mourato, M. Santos, and F. Birra. Automatic level generation for platform videogames using genetic algorithms. page 8, 11 2011.

[10] C. Reuter, V. Wendel, S. Göbel, and R. Steinmetz. Game design patterns for collaborative player interactions. In *DiGRA*, 2014.

[11] J. B. Rocha, S. Mascarenhas, and R. Prada. Game mechanics for cooperative games. 2008.

[12] 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*, CHI '10, page 253–262, New York, NY, USA, 2010. Association for Computing Machinery.

[13] N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[14] J. Zagal and J. Rick. Collaborative games: Lessons learned from board games. *Simulation & Gaming - Simulat Gaming*, 37:24–40, 03 2006.