Generating basic structures for Angry Birds via free-form evolution

First Author Name[1][a], Second Author Name[1][b] and Third Author Name[2][c]

[1]*Place Holder, XYZ University, My Street, MyTown, MyCountry*
[2]*Place Holder, Main University, MySecondTown, MyCountry*
{f_author, s_author}@ips.xyz.edu, t_author@dc.mu.edu

Abstract: In this work, we present an original method based on evolutionary algorithms for generating basic structures for the physics-based game Angry Birds, with the ultimate objective of creating Angry Birds levels with the minimum number of constraints. We must generate structures that remain stable under gravity, in order to be playable in the game. We have designed and implemented an open source evolutionary computation library, with a fitness function designed to first, avoid if possible the an actual simulation of a game, which is time-consuming, and, then, to take into account the different ways in which a structure is not sound and consider how to achieve that soundness. We have experimented with different representations and evolutionary operators. In order to evaluate the algorithm, we carried out six experiments, obtaining a wide variety of stable structures. These structures are the basis for generating levels that are aesthetically pleasing as well as playable.

1 INTRODUCTION

Angry Birds is a multiplatform video game created in 2009 by the Rovio Entertainment Corporation. Each level of the game consists of a collection of structures made out of blocks in which comic *pig* characters are hiding; the player has to fire from a slingshot different *bird* characters, each having different abilities and moods. The objective of the game is to destroy all the pigs by knocking down the structures or just hitting them directly. The game relies on gravity to create interesting puzzles by closely resembling the dynamics of real-world structures. From the Procedural Content Generation (PCG) perspective, the challenge is to generate stable structures that are robust enough to take more than a single hit before crumbling to the ground. Even before that, they need to stand on their own as the level starts.

The objective of this work is to design an algorithm capable of generating basic game structures for *Angry Birds* and eventually generating levels using these components. We can break it down to the following, more concise objectives:

- Design a Search-Based PCG algorithm (SBPCG) to exploit the expressiveness and variability of EAs.
- Modify the simulator to extract data from the execution to evaluate the generated game levels.
- Generate stable structures under gravity.

The main focus of the paper is on the first two objectives. First, to produce free-form structures that do not collapse, that is that they are valid. Also, to extract information from the generated structures to be able to evaluate them without needing to run a costly simulation. We will target replayability by emphasizing the generation of free-form structures. Free-form, in this case, means that structures do not follow a preconceived pattern. If all levels follow a simple or repetitive pattern, users could be bored and disengage. Our goal is to avoid this by creating diverse structures whose form is only constrained by gravity.

Search-based Procedural Content Generation (SBPCG), is a type of a *generate-and-test* approach to PCG which is usually tackled with Evolutionary Algorithms(EA) (Togelius et al., 2010). The challenges faced by SBPCG are not far from those found in EAs; since they are search methods, they can be a good option to perform this kind of systems.

The main focus of the paper is on the first two ob-

---
[a][iD] https://orcid.org/0000-0000-0000-0000
[b][iD] https://orcid.org/0000-0000-0000-0000
[c][iD] https://orcid.org/0000-0000-0000-0000

jectives. First, to produce free-form structures that do not collapse. Also, to extract information from the generated structures to be able to evaluate them without needing to run a costly simulation. We will also target replayability by emphasizing the generation of free-form structures. Free-form, in this case, means that structures do not follow a preconceived pattern. If all levels follow a simple or repetitive pattern, users could be bored and disengage. Our goal is to avoid this by creating diverse structures whose form is only constrained by gravity. However, free-form also implies that structural integrity is not guaranteed, and as structures have a realistic gravity, they could collapse at the beginning of a level. We need to incorporate this factor into the *fitness function*. We will be searching *valid* structures, whose playability has to be optimized. In this paper, we do not address other aspects of level design like aesthetics, the ability of players or gameplay. For instance, the search gives no advantage to symmetric structures or challenging levels.

In this paper, we will use a *direct fitness function* (Togelius et al., 2010); in this kind of functions, we extract measurable features from the generated content and map them to a fitness score. However, this fitness function is a time-consuming task since it involves the generation of a graphic representation of the structure and the simulation of the falling motion. If we have to evaluate every single individual in the population, we will not be able to cover enough of the free-form search space to find a good enough solution. So we must minimize the actual number of structures that are simulated by applying heuristics to the data structure and assigning it a fitness even before the simulation.

This paper is organized as follows: in the next section, we present the state of the art in this type of level generation, as well as its relation to the problem of generating structurally sound structures. Our proposed method for generating Angry Birds levels is described next in Section 3. Next experiments are presented in section 4. We present our conclusions in 5.

2 Background and State of the Art

PCG used for video game levels is relevant in international artificial intelligence competitions, such as Super Mario Level Generation (Shaker et al., 2012), General Video Games (Khalifa, 2018; Khalifa et al., 2016), or recently, for Angry Birds (Stephenson et al., 2018). This work is related to works presented in previous editions of the competition. The focus of the latest edition (Stephenson et al., 2018) was on find-

ing entertaining levels. Fun was the main factor in the evaluation of proposals; secondary factors were creativity and difficulty. Six entries participated, of which J. Yuxuan et al., were able to generate random quotations with different components of a level; J. Xu et al. generated levels that look like pixel images. A third approach (by C. Kocaogullar) translated music patterns to generate structures. The winner was Iratus Aves, a new iteration of the work by M. Stephenson and J. Renz (Stephenson and Renz, 2017; Stephenson and Renz, 2016), which follows *constructive method*. In this work, the likelihood of selecting a certain block is given by a probability table, which was tuned using an optimization method. Then, blocks are placed following a tree structure.

A constructive method ensures local stability, but not global stability, which must be evaluated once the whole structure is completed. One problem with this and other constructive approaches is that the variety of structures created is going to be relatively small; monotony leads to boredom, decreasing playability. On the other hand, generated structures are guaranteed to be structurally sound, and constructive approaches are generally faster than search-based procedures.

Search-based approaches propose a good alternative to deal with these limitations. Thus, Lucas Ferreira and Claudio Toledo (Ferreira and Toledo, 2014) presented a solution – ranked 4th on the last competition – based on SBPCG, which uses a Genetic Algorithm (GA) and a game clone named *Science Birds* developed to evaluate the levels, the same one we use in this paper.

In a GA individuals correspond to levels, each represented by an array of columns. Each column is a sequence of blocks, pigs and predefined compound blocks, using an identification integer. This representation also includes the distances between different columns. The population is initialized randomly following a probability table which defines the likelihood of a certain element being placed in a certain position inside a column. This implies that a shape is chosen beforehand, once again to guarantee stability, but decreases playability by generating structures whose only differences are which blocks are placed on top of which. Levels are evaluated executing them in the simulator and checking their average stability, considering the speed of every block when erected – which must tend to be zero for having a stable structure –. The authors designed specialized crossover and mutation operators, aiming to maintain some consistency in the new solutions generated. However, this work proposes a different approach: *free-form evolution*. If we look outside the domain of game de-

velopment and focus on structural optimization in architecture, there are several proposals using search-based algorithms. We can find a metaheuristic called Cuckoo Search (Gandomi et al., 2013) which performance was tested with structural optimization. However, this optimization is heavily parametrized and we are looking for the evolution of structures that do not follow a predefined pattern. Another approach for structure design is using Generative Grammatical Encodings (Hornby and Pollack, 2001) where L-system and its production rules are considered individuals. This method increases the number of generated patterns, but still restrains the formation of disjoint structures, for instance, a defensive tower before a simple pigsty in our context.

We aim at following a realistic structure generation approach, without constraining it to a fixed form, thus advancing the state of the art by allowing the creation of Angry Birds levels having any arrangement. The next section will describe how we characterize this problem and our approach to it. In our previous paper (Calle et al., 2019) we explored this approach and found as one of its shortcomings the fact that the evaluation of generated structures through the simulator was lengthy and didn't leave the algorithm enough time to explore the search space. In this paper, we try to tackle that problem, as well as take additional steps to increase the complexity of generated structures.

3   Problem Description

Science Birds is nowadays the main open source Angry Birds simulator. It was developed by Ferreira and Toledo (Ferreira and Toledo, 2014), and is available from GitHub. We used it as a starting point; however we had to patch it for producing usable output and automate its work. The modified version is available on GitHub `https://github.com/Laucalle/ScienceBirds`. It produces output containing the position and average magnitude of the velocity of each block that was not broken after the simulation. It can be run from beginning to end without user interaction and minimizes the amount of time spent on simulating each level.

In later experiments we will use Box2D (`https://box2d.org`), the Physics engine, written in C++, originally used for the actual Angry Birds game, to avoid launching the whole game which adds overhead to the fitness evaluation. In this simulation we do not have the resistance of the blocks implemented but we can test the level stability much more efficiently, since there is no overhead computing things unrelated to Physics, such as the graphic interface itself. This

means we cannot penalize levels per broken block.

Once the simulator that is going to be used to measure fitness is ready, we have to design the fitness function. As obvious as it might seem, the main feature of a stable level is that it is not in motion, so it seems reasonable to evaluate its whole stillness as opposed to its speed – considering every single block –.

$$fitness_{ind} = \frac{1}{|V|} \sum_{i=0}^{b} V_i + P_{broken} \cdot (b - |V|)$$

In this version, the game engine provides the average magnitude of velocity for each block. This set is noted as V, with $|V|$ being the cardinality of the set. b is the number of blocks in an individual and it can differ from the cardinality of V since collapsed blocks are not tracked. The number of broken blocks is $b - |V|$ and it is multiplied by a penalization factor, since a level whose blocks break without user interaction would not be considered valid. This happens when a block free-falls from a certain height or collides with a falling object. P_{broken} is set to 100 since objects in a level do not usually reach that velocity, therefore it will separate non-valid levels from potentially good ones.

In the sixth experiment this fitness function is changed to the following function after observing the results for the previous experiment, al of them described in section 4:

$$fitness_{indV2} = \max(V)$$

Either way, simulating a level is quite time consuming, on the order of seconds using the game, which makes it almost unfeasible for our purpose, so we decided to take additional factors on the fitness so that not all levels are actually simulated. For that reason, before testing a level, there are some indicators, like distance to the ground and number of overlapping blocks, that a level would not be a valid solution and thus simulation can be skipped.

If the object closest to the ground is far above, it is very likely that it, along with all the others blocks above, will collapse from the impact. Levels that have all their blocks higher than a certain threshold will not be simulated by the engine. The threshold used is 0.1 in game units and the penalty applied to the distance in order to compute the fitness value is 10:

$$f_{distance} = \begin{cases} P_{distance} \cdot D_{lowest}, & \text{if } D_{lowest} > threshold \\ 0, & \text{otherwise} \end{cases}$$

The other measure is the number of overlapping blocks. The separating axis theorem (Ericson, 2004) determines if two convex shapes intersect. It is commonly used in game development for detecting collisions. A level with blocks that occupy the same

space is not likely to be stable, as the Unity Engine underlying the simulator will solve the issue moving the blocks until there is no collision, but this is done by Unity proprietary code and it is not possible to know what it does. So, a penalty is applied and the level is not simulated either. In this case it is $f_{overlapping} = P_{overlapping} \cdot N_{overlapping}$ where the first factor is a penalty set to 10 and the second is the number of blocks that overlap with each other.

In some of the late experiments we will substitute the $f_{distance}$ with the gap in the Y-axis. We project all blocks on the Y-axis and calculate the range of values for the Y coordinate that are not covered inside the feasible range. This is treated the same way as $f_{distance}$ (same penalization and threshold) and we call it f_{Y-axis}:

$$f_{Y-axis} = \begin{cases} P_{distance} \cdot Pr_{Y-axis}, & \text{if } Pr_{Y-axis} > threshold \\ 0, & \text{otherwise} \end{cases}$$

If both $f_{distance}$ and $f_{overlapping}$ are 0 then the level is suitable for simulation and fitness is calculated as $fitness_{ind}$. This would be considered *overpenalization* but exploring unfeasible regions entails a serious overhead that we need to minimize (Runarsson and Yao, 2003). On the other hand, levels with multiple blocks broken during the simulation are not feasible either but running the simulation is necessary. In this case, the penalization does not prevent the region to be explored.

Since one of the objectives of this work is to explore the expressiveness and variability of SBPCG, it seems reasonable to use a flexible representation. We will test these solutions to allow a less directed search than previous solutions while keeping a simple representation.

Individuals are composed of a list of blocks; platforms, TNT boxes or pigs are not considered in this paper, since we are focused on the generation of structures. These building blocks have the following attributes:

- Type: there are eight regular blocks that can be placed in the level with distinct shapes or sizes; they are represented as an integer between 0 and 7.

- Position: coordinates *x* and *y* of the centre of the block in game units.

- Rotation: rotation of the block in degrees. Four different rotation angles are considered: 0, 45, 90 or 135 degrees represented as integers between 0 and 3.

- Material: three types, which determine the durability of the block. However, this does not affect

their stability, so it will remain constant for now as *wood* material.

Using this representation a gene representing a single block will be formed by two integers and two floating point numbers.

Individuals are a collection of genes, in the same way a level is a collection of building blocks. The number of blocks is variable and the order in which they are listed is not important.

The fitness of the worst individual that has been tested in game is stored, so that the value of not tested levels is always above —it is a minimization problem— the in-game tested levels; the starting point for fitness of such individuals is the worst in-game score.

This penalization is calculated using the distance of the lowest block to the ground, which can be easily obtained, and the number of blocks that collide. This requires a bit more of computation, so it will be stored and set in the initialization of the individual. When a gene is modified, the number of overlapping blocks is recalculated for that specific change.

Considering all of the above, the chromosome object is composed by:

- A list of genes.

- A fitness value.

- A penalty (set to `False` for in-game evaluated levels).

- Number of overlapping blocks (calculated).

Initialization is done randomly, with each individual having a random number of genes, which are initialized by several methods:

- Random: selects a random number for each attribute of the gene.

- No Overlapping: also selects a random number but the gene is only added to the chromosome if it does not overlap with an already existing gene.

- Discrete: selects a random number for type and rotation, but the position must be multiple of the dimensions of the smallest block (blocks will be aligned).

- Discrete without overlapping: it combines the second and third initialization method.

- Discrete with a set of pre configured blocks: first it includes a set of blocks, and then adds blocks following the third method until it reaches the desired number of blocks. The configurations used are the compound blocks found in (Ferreira and Toledo, 2014).

Candidates for reproduction are selected using tournaments. Two individuals are chosen from the population and the best will be a parent in this generation. This is repeated until a certain percentage of pairs have been reached. It is important to note that individuals chosen are not removed from the population and therefore they can appear several times in the list of parents.

Once the parents have been selected, we implement two different methods of combination:

- Sample Crossover: gives a single individual per parent pair. It takes all genes from both parents—excluding genes that are repeated—and randomly takes a number of them to create the new individual. The number of blocks is the minimum between the maximum number of blocks allowed, the mean of the two parent individuals and the number of distinct genes.

- Common Blocks: produces two individuals. The common genes to both parents are passed on to both children. The remaining genes are randomly distributed to each child, half to one and half to the other.

There are four different mutations:

- Rotation: rotation is represented as an integer (it is discretized), so it adds or subtracts one to the current value.

- Type: similarly to rotation mutation.

- Position X: a real value between 0 and 1—excluding 0—is added or subtracted from the value of the position X.

- Position Y: same as position X mutation, for position Y.

They are all applied to random members of the population.

The new generation is selected using an elitist strategy. Best individuals in both the old population and their offspring pass on the next generation, maintaining the size of the population.

The information that describes a level can be too complex to have a binary representation as pure genetic algorithms suggest, so the framework should be flexible enough to support complex data structures. This prevented us from using other frameworks and therefore a new framework was created. The source code can be found on GitHub at `https://github.com/Laucalle/AngryBirdsLevelGenerator`.

In order to evaluate the different options and check if they meet our objective, we performed a series of experiments presented in next section.

	Time(h)	σ	G	σ
E1	0.89	(0.59)	100.0	(0)
E2	1.002	(1,97)	155.087	(240.56)
E3	1.76	(0.6)	76.625	(42.3)
E4	5.03	(1.46)	365.929	(158.09)
E5	0.099	(0.1)	121.2	(96.89)
E6	0.788	(0.124)	1000.0	(0)

Table 1: Summary of the execution of the last generation in 15-20 runs for each experiment. 40 runs for E5 and E6. G: number of generations, E: experiment number

	Best	σ	Avg	σ	Worst	σ
E1	61.334	(133.02)	383.701	(106.14)	510.515	(133.04)
E2	110.66	(142.21)	327.547	(238.33)	367.895	(260.83)
E3	0.0015	(0.003)	0.54	(0.24)	0.828	(0.34)
E4	0.0018	(0.003)	0.203	(0.068)	0.2997	(0.1)
E5	1.249	(1.257)	1.276	(1.231)	1.288	(1.219)
E6	1.031	(0.853)	1.27	(0.834)	1.328	(0.819)

Table 2: Summary of the results of the last generation in 15-20 runs for each experiment. 40 runs for E5 and E6. G: number of generations, E: experiment number

4 Experiments and Results

We set out to evolve free-form structures, but we need to test, one by one, the different parts of our algorithm: fitness function, some evolutionary algorithm parameters and the genetic operators. In order to do so, we performed a set of experiments to test our hypotheses, whose result is shown in Table 1 shows an overview of the results.

4.1 Baseline experiment

The premise of the first experiment is that our basic EA should be able to minimize the movement of the blocks placed on the level and the flexibility of the representation should allow variety in the structures. This will be used as a baseline.

The EA in this experiments uses initialization with the discrete method described earlier, basic sample crossover, all four mutations, elitist replacement and tournament selection using the parameters are in table 3.

The results suggest that the hypothesis was not correct. The average best solution has a fitness value of 61.334 (as shown in Table 2) which indicates that probably most levels have blocks falling (and even breaking) when loaded. The standard deviation of this measure is 133.0209 which implies that while some executions performed poorly, some others may be good. Even the best levels have blocks that break after loading so they would not be valid. However,

Table 3: Parameters used in the first experiment

Population size	100
Number of generations	100
Percentage of parents	0.5
Percentage of type mutations	0.5
Percentage of rotation mutations	0.5
Percentage of axis x mutations	0.5
Percentage of axis y mutations	1

Figure 2: One of the results for 4.1: Fitness = 9.021, G = 100

we can tell that there is variety in the structures, since they clearly differ from each other.

In the experiment the only termination condition was reaching the maximum number of generations. However, looking at the results, it seems that the execution ended before the population stabilized or converged. Since mutation percentages are high it is normal that convergence where every single individual is the same one, is not reached. However, it could be possible that the population is stable, where every child has a greater fitness value than its parents, therefore no new members are allowed. If there are no new individuals and the population is completely *stuck*, the fitness value of the worst individual should be the same over several generations. In Figure 1 we can see that this is not the case in average. Although some populations do remain the same for several generations close to the maximum, most of them do not.
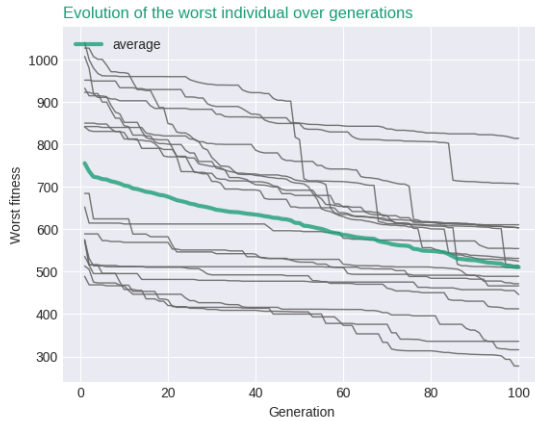

Figure 1: In grey, different executions of the first experiment (E1)

Solutions found in this experiment contained just a few blocks on the ground, occasionally those were stacked as seen in figure 2. Most executions from the previous experiment reached the maximum number of generations without stabilizing or converging. This means the EA may need a larger number of generations to fully evolve a solution. This is the hypothesis for the second experiment, which we describe next.
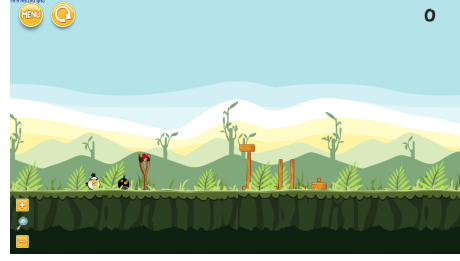
4.2 Changing termination conditions

The main change in this implementation is the addition of two new stop conditions: being 10 generations without changes (stable population) or best fitness value below 0.01. The set of operators is the same as the previous one, and the parameters remain unchanged except for the *Number of generations* which is equal to 1000. The increase in the number of generations allows each run to fully develop their individuals. This would not be feasible if we had to simulate all the individuals, but thanks to the fitness function and the new stop conditions we can afford this number of generations.

Although the best levels obtained with this experiment are better than those evolved in the first experiment, the bad solutions have a really high fitness value. An example of one the best levels obtained is shown in figure 4. In table 2, we can see that average fitness of levels produced with this version of the EA is worse than the ones generated in the first experiment. It suggest that populations can be stuck for many generations before making any type of improvement. Any of the generated levels have a fitness below 0.01 or reached maximum number of generations, which means the termination criteria that stopped the evolution was that the population was stable, without any new individuals added for 10 generations.

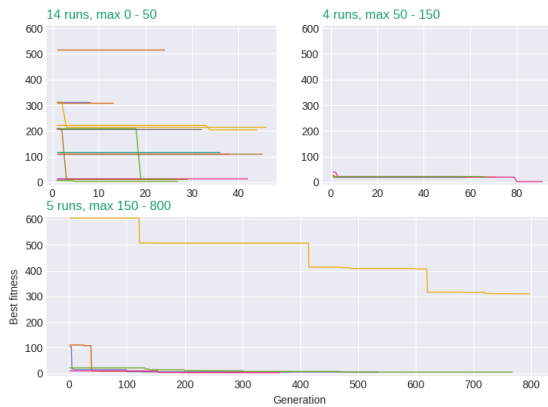Figure 4: One of the solutions from 4.2: Fitness = 5.176, G = 767

Figure 3: Best individual evolution for all executions, grouped by number of generations

Figure 3 represents the evolution of the best individual of each execution. Most of them have no more than 50 generations, therefore the hypothesis for this experiment is not correct. Short evolutions show that the best individual at initialization is very similar to the last one. Slight improvements may be achieved by small mutations but it seems difficult for new generations to outperform previous ones. We can appreciate that significant improvements are most common in those executions with a poor initial population. Even the ones with several hundreds of generations struggle to improve the initial population.

Our hypothesis that the number of generations could be the main factor is then not correct; it is more likely that there this EA is biased towards exploration rather than exploitation. The genetic operators are failing to create new individuals that inherit good traits from their parents. A new crossover operator could shift the focus to exploitation, and we will examine this in the next experiment.

4.3 Improving exploitation using a better crossover operator

For the third experiment, the change introduced is in the crossover operator used which was previously described as *Common blocks* crossover. The rest of the operators as well as the termination criteria are kept the same.

Table 2 shows that the results have radically improved as the average fitness of the best solutions drops to 0.0015, a decrease of almost 100%. Additionally, it took less generations in general to reach those results. However, executions took longer on average, which makes sense given that a greater number of individuals would have been simulated. The average fitness of the population and the worst individual have similar values now, which suggest that in most executions the population did converge.

The levels are stable and the blocks do not fall when loading the level, but it is arguable that they would be considered structures, since most of them consist in a few blocks spread about on the floor. The average amount of blocks is 6.26, which is really close to the minimum amount of blocks allowed. However, given the proposed fitness function, it is completely logical that the evolution leads to this kind of arrangements. The more objects placed on the level, the more likely the individual is to not meet the requirements imposed by the constraints. It also makes sense to place objects near the ground, instead of one on top of the other.

The fitness landscape a fitness function creates is difficult to assess, but in this case it is clear that search goes on the direction of minimizing the number of blocks so that the number of blocks actually falling is zero. The created structures are not very interesting, though, that is why we propose some changes in the next experiment.

4.4 Changing the minimum number of blocks per individual

The only parameter that was changed for this experiment is the minimum number, which went from 5 to 10.

The first thing to notice in these experiments (Table 1) is the increase of the average execution time: it went from 1.76 hours in the third experiment to 5.03 hours in this one. The time spent running the simulation for each population in this experiments and in the previous one should be similar. However, the number of actual executions of the simulator, which only kick in if there is actually some block on the floor and there is no overlap, drastically increased too. There
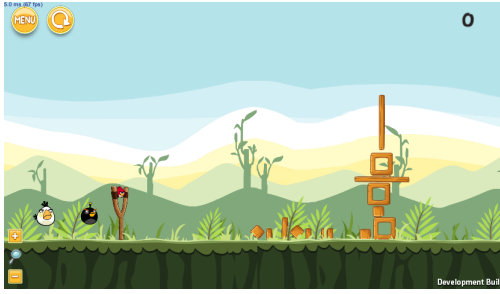
Figure 5: One of the solutions from 4.5

Figure 6: One of the solutions from 4.6

is no doubt that placing at least ten objects in a structure that does not collapse is more difficult than placing just five. The average best fitness value increased slightly, while the average and worst values are lower. This suggest that the latest generations of this EA are less diverse than those from the third experiment.

However, the results are more interesting visually. We can find some blocks being stacked together, not only lying on the ground as in previous experiments.

4.5 Removing game and penalizing gaps in Y-axis

The main problem with the previous experiments was the time needed to load the Science Birds simulation environment and run the levels, which needed several seconds for loading and obtaining results. So the main objective of this experiment was to find a way to get rid of the in-game simulation. In order to do that, we will use a Physics engine called Box2D (Catto, 2011) that was used in the development of the original game. Since game physics do not usually resemble real world physics we adjusted the parameters so this simulation an the game behave in the same way. As we can see in table 1 the execution time drastically drops from 5 hours to less than 20 minutes on average, even running more generations in the process.

This opened the way for performing more operations on the individuals. In this case we chose to penalize not only the distance to the ground but also the *gaps* in the Y-axis, which will make objects drop and maybe break. This will encourage individuals to grow vertically and not only horizontally like in previous experiments. This changes the fitness function, so we will have to compare by the actual obtained structure, one of which is shown in Figure 5.

In general, this penalization of gaps creates a faster path to stable structures. Still, this path leads to mostly flat structures with some higher block in unstable positions, which are structurally solid, but not interesting. One of the stable results is shown in figure 5.

4.6 Changing the evaluation function

Observing results from the previous experiment we realized that what evolution found was that laying many blocks on the ground was enough to get a high fitness: the average speed was decreased and it will place unstable blocks to cover gaps in Y-axis. In order to correct this behaviour we changed the fitness function to take into account the fastest moving object:

$$fitness_{indV2} = \max(V)$$

Additionally, we initialized levels including one of a list of pre configured blocks in addition to the random initialization used until now.

This makes the fitness value depends on just one gene, although it can be a different one each time. The improvement of solutions to find acceptable ones slowed down again, with a different fitness function we cannot compare the fitness value with the rest of the experiments.

5 Conclusions and Future Work

This paper was developed with the main objective of implementing a competitive entry for the Angry Birds level generation contest, and a set of sub-objectives in mind: exploring the expressiveness and variability of SBPCG with evolutionary techniques, adapting the game to extract data from execution and producing stable structures under gravity, in a simulated environment.

For this aim we have implemented an Evolutionary Algorithm able to optimize level structures to meet some criteria, being the main one the stability of the constructions. Considering this, the method studied was sufficiently general and flexible to draw some conclusions about the topic. SBPCG methods are a potential good solution to offline content generation but it requires a great amount of problem-specific knowledge. Like any other form of creative work, the biggest issue may be how to measure how

good, creative or enjoyable is the piece. The more rules the author adds, expressiveness starts to get lost as the results are variations of the same idea. However, it is crystal clear from the experiments run in this paper that a lack of knowledge will lead to unexpected outcomes. In our case, the fitness function used the stability of the structure and only considered other features— overlapping blocks and distance to the ground— to ensure the levels would be valid. In experiment 4.5 and 4.6 height was also taken into account.

The second objective, adapting the game to extract data from execution, was certainly achieved. It was also a basic requirement to proceed with the rest of them. The game does provide the data, as long as the input is correctly structured. In order to conduct our experiments the Angry Birds simulator (Science Birds) has been adapted to our necessities, yielding now some other information required to evaluate the individuals of the implemented EA. As this was a bottleneck, experiments 4.5 and 4.6 used only a physic simulation, leaving behind the actual game. It would be interesting to obtain other kind of data from both simulations, such as the height of the structure and, eventually, its resistance to bombardment by angry birds. However, this is left as future work.

Producing stable structures under gravity was the third objective and the closest to the ultimate one. That objective was effectively achieved, but the consequence of evolving in a path of minimum movement or maximum stability results in structures that are close to the ground and do not have more than a few floors , which could not be very exciting for the players. The main issue is, then, how we evaluate the levels. In fact this is a matter of how we define what an Angry Birds' level *is* and if that definition matches the fitness function. How this definition of level plays with the paths of evolution is also a problem. Since we define as level as a structure that is stable, evolution will maximize stability finding, as in the beginnings of architectural practice, squat structures that are neither aesthetically pleasing nor playable, although undeniably sturdy and stable. We have been successful in, evolving free form structures, to find these type. But once we get there, we need to go into a different evolution mode that takes into account several features as a multiobjective optimization problem: stability is the first, but we can sacrifice a bit of stability for height or some other aesthetic quality.

To conclude on an optimistic tone, this work provides an interesting insight into the SBPCG, through the completion—and failure—of the goals we set out to achieve at the beginning.

In order to improve the results of the method, different constraints could be expressed as multiple objectives. Overlapping blocks and velocity could be treated as minimization objectives and height as a maximization one.

If we pay attention at the stages of evolution in this work, there is also room for improvement in the genetic operators. For example, the initialization produces a small amount of valid individuals which suggested that an elitist strategy for selection would work best. However, new experiments will help to better balance exploration and exploitation. An interesting addition would be to add *building* operators that pile blocks on structures that are already stable.

6 Acknowledgements

REFERENCES

Calle, L., Merelo Guervós, J. J., García, A. M., and García Valdez, J. M. (2019). Free form evolution for angry birds level generation. In Kaufmann, P. and Castillo, P. A., editors, *Applications of Evolutionary Computation - 22nd International Conference, EvoApplications 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24-26, 2019, Proceedings*, volume 11454 of *Lecture Notes in Computer Science*, pages 125–140. Springer.

Catto, E. (2011). Box2D: A 2D Physics engine for games.

Ericson, C. (2004). *Real-time collision detection*. CRC Press.

Ferreira, L. and Toledo, C. (2014). A search-based approach for generating angry birds levels. In *Computational intelligence and games (cig), 2014 ieee conference on*, pages 1–8. IEEE.

Gandomi, A. H., Yang, X.-S., and Alavi, A. H. (2013). Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. *Engineering with computers*, 29(1):17–35.

Hornby, G. S. and Pollack, J. B. (2001). The advantages of generative grammatical encodings for physical design. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 600–607. IEEE.

Khalifa, A. (2018). The general videogame ai competition - level generation track. https://github.com/GAIGResearch/gvgai/wiki/Tracks-Description\#level-generation-track. Accessed: 2018-11-05.

Khalifa, A., Perez-Liebana, D., Lucas, S. M., and Togelius, J. (2016). General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 253–259, New York, NY, USA. ACM.

Runarsson, T. P. and Yao, X. (2003). Evolutionary search and constraint violations. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 2, pages 1414–1419. IEEE.

Shaker, N., Togelius, J., and Yannakakis, G. (2012). Mario ai championship - level generation track. `http://www.marioai.org/LevelGeneration`. Accessed: 2018-11-05.

Stephenson, M. and Renz, J. (2016). Procedural generation of complex stable structures for angry birds levels. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE.

Stephenson, M. and Renz, J. (2017). Generating varied, stable and solvable levels for angry birds style physics games. In *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pages 288–295. IEEE.

Stephenson, M., Renz, J., Ferreira, L., and Togelius, J. (2018). 3rd angry birds level generation competition. `https://project.dke.maastrichtuniversity.nl/cig2018/competitions/\#angrybirds`. Accessed: 2018-11-05.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2010). Search-based procedural content generation. In *European Conference on the Applications of Evolutionary Computation*, pages 141–150. Springer.