PROCEDURAL MAP GENERATION FOR A RTS GAME

Raúl Lara-Cabrera, Carlos Cotta and Antonio J. Fernández-Leiva

Department "Lenguajes y Ciencias de la Computación"
University of Málaga
Louis Pasteur, 35, 29071, Málaga – Spain
{raul,ccottap,afdez}@lcc.uma.es

ABSTRACT

Procedural content generation (PCG) is the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces. This methodology brings many advantages to game developers, such as reduced memory consumption. In this paper we introduce a procedural map generator for a real-time strategy (RTS) game. The main component of this generator is a genetic algorithm devoted to create and evolve balanced maps, i.e. maps where no player has any map related advantage with respect to other players. The selected RTS game is called *Planet Wars* and it was used in the *Google AI Challenge 2010*. It is a space conquest game whose objective is to take over all the planets on the map.

INTRODUCTION

This paper introduces a map generation method for a RTS game. This method can be classified as a procedural content generation method (PCG). PCG refers to creating game content automatically, through algorithmic means. This content refers to all aspects of the game that affect game-play other than non-player character (NPC), such as maps, levels, dialogues, characters, rule-sets and weapons. PCG is interesting for the game developing community due to several reasons, such as reduced memory consumption and the saving in the expense of manually creating game content. Our map generation method can be categorized (using the taxonomy proposed in Togelius, Yannakakis, Stanley & Browne 2011) as an off-line method that generates necessary content, using random seeds and deterministic generation and following a generate-and-test schema.

Procedural content generation has been used in many well-known video-games. *Borderlands* Gearbox Software 2009 uses a PCG system to create weapons and items, which can alter their firepower, rate of fire, and accuracy, add in elemental effects such as a chance to set foes on fire or cover them in burning acid, and at rare times other special bonuses such as regenerating the player's ammo. PCG system is also used to create the characteristic of random enemies that the player may face. Another example of a game that uses PCG is *Minecraft* Mojang 2011, a sandbox-building game with an infinite map which is expanded dynamically. *Spore* Maxis 2008 is a god game simulation that contains multiple levels of play, from starting as a multi-celled organism in a tide pool, up to exploring a dynamically generated universe with advanced UFO technology. The music of the game is also procedurally generated.

From an academic point of view, there are several papers related to procedural map generation. In Togelius, De Nardi & Lucas 2007 the authors designed a system for offline/online generation of tracks for a simple racing game. A racing track is created from a parameter vector using a deterministic genotype-to-phenotype mapping. A search-based procedural content generation (SBPCG) algorithm for strategy game maps is proposed in Togelius, Preuss & Yannakakis 2010 from a multi-objective perspective. A multi-objective evolutionary algorithm is used for searching the space of maps for candidates that satisfy pairs of these multiple objectives. Another search-based method for generating maps is presented in Togelius, Preuss, Beume, Wessing, Hagelback & Yannakakis 2010. In this case, the maps are generated for the game *Starcraft* Blizzard Entertainment 1998. Frade et al. have introduced the idea of terrain programming, namely the use of genetic programming to evolve playing maps for video-games, using either subjective human-based feedback Frade, de Vega & Cotta 2008, Frade, de Vega & Cotta 2009 or automated quality measures such as accessibility Frade, de Vega & Cotta 2010*a* or edge-length Frade, de Vega & Cotta 2010*b*. In Mahlmann, Togelius & Yannakakis 2012 the authors describe a search-based map generator for an abstract version of the real-time strategy game *Dune 2*. Map genotypes are represented as low-resolution matrices, which are then converted to higher-resolution maps through a stochastic process involving cellular automata.

In the next section we describe the RTS game that has been used in the experiments. Then, we describe a procedural map generator and a genetic algorithm that creates and evolves balanced map. Right after this description, there is a section where we report the results we

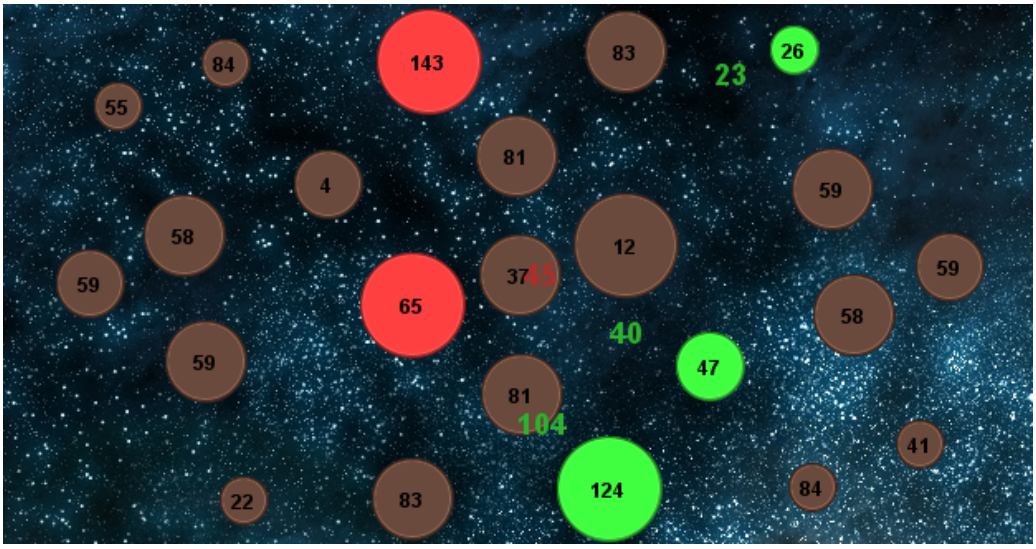

Figure 1: A screenshot of Planet Wars

have obtained from the experiments. Finally, we present our conclusions and future work.

GAME DESCRIPTION

Planet Wars is a real-time strategy (RTS) game based on *Galcon* and used in the *Google AI Challenge 2010* (a screenshot is shown in figure 1). It is set in outer space and its objective is to take over all the planets on the map, or alternatively eliminate all of your opponents ships. A game of Planet Wars takes place on a map which contains several planets, each of which has some number of ships on it. Each planet may have a different number of ships. The planets may belong to one of three different owners: you, your opponent, or neutral. The game has a certain maximum number of turns. The game may end earlier if one of the players loses all his ships, in which case the player that has ships remaining wins instantly. If both players have the same number of ships when the game ends, its a draw. On each turn, the player may choose to send fleets of ships from any planet he owns to any other planet on the map. He may send as many fleets as he wishes on a single turn as long as he has enough ships to supply them. After sending fleets, each planet owned by a player (not owned by neutral) will increase the forces there according to that planets growth rate. Different planets have different growth rates. The fleets will then take some number of turns to reach their destination planets, where they will then fight any opposing forces there and, if they win, take ownership of the planet. Fleets cannot be redirected during travel. Players may continue to send more fleets on later turns even while older fleets are in transit. Despite players make their orders on a turn-by-turn basis, they issue these orders at the same time, so we can treat this game as a real-time game.

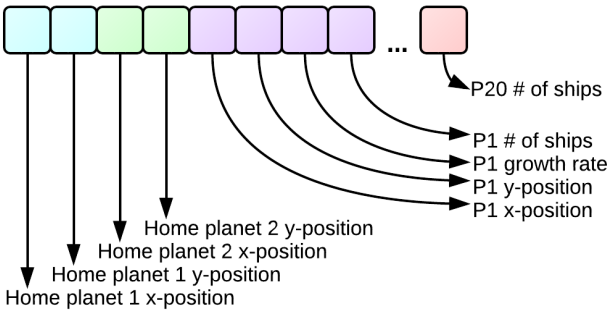

Figure 2: Structure of the individual

Maps have no particular dimensions and are defined completely in terms of the planets and fleets in them. They are defined in plain text files, with each line representing a planet or a fleet. Planet positions are specified relative to a common origin in Euclidean space. The coordinates are given as floating point numbers. Planets never move and are never added or removed as the game progresses. Planets are not allowed to occupy the exact same position on the map. The owner of a planet can be neutral, player 1, or player 2. The number of ships is given as an integer, and it may change throughout the game. Finally, the growth rate of the planet is the number of ships added to the planet after each turn. If the planet is currently owned by neutral, the growth rate is not applied. Only players can get new ships through growth. The growth rate of a planet will never change. It is given as an integer.

PROCEDURAL MAP GENERATOR

In this section, we present a search-based procedural map generator that is capable of generating balanced maps for the real-time strategy game *Planet Wars*. It is composed of two systems, a genetic algorithm responsible for generating and evolving maps and a system responsible for playing Planet Wars games and evaluating the maps. The evaluator is a tool developed by *Google* for the *Google AI Challenge 2010*. It has been developed using Java and is a console application. It runs a customizable game between various players and generates a game trace. The game can be viewed with a visualizer (included in these tools) which reads this game trace. We have created a script that calls the evaluator with a specified map and stores the game trace to a file. Later, this file is processed to compute the fitness of the generated map. We have used a Java-based evolutionary computation research system, called ECJ[1], for constructing the genetic algorithm (a review of this system can be found in White 2012). It supports multithread evaluation and breeding, a master-slave architecture, island models, and even experimental support for GPGPU through a third-party extension. It is easily configurable because of it simple text-based parameter files. Its implementation in Java makes ECJ very portable. Unless ECJs graphical user interface (GUI) is needed, ECJ is self contained. The integration of networking and serialization support that Java provides makes developing new parallel architectures and checkpointing methods much easier than starting from scratch or using a third-party library.

The genetic algorithm generates maps with 20 neutral planets and two starting planets (one for each player), i.e. maps with 22 planets. The proposed genetic algorithm follows a generational scheme with elitism (the best solution always survive). As described on the previous section, every planet has five properties: x-position, y-position, owner, growth rate and number os ships. To obtain a balanced map, we have fixed the growth rate of the two starting planets. Planets' owners have been also fixed, so we got finally 84 parameters (4 for every neutral planet, and 2 for every starting planet). Each individual of our genetic algorithm is made of these 84 parameters, grouped into a vector of floating point numbers in the range between 1 and 5. The first two parameters are the x and y position of the home planet for player 1, while the next two parameters correspond to the position of the home planet for player 2. Then, there are 20 groups of 4 parameters, one group for each neutral planet, whose parameters are the x position, y position, growth rate and number of ships respectively (see figure 2).

The algorithm (see table 1) uses a population of 40 individuals on each generation with a runtime of 100 gen-

[1] `http://cs.gmu.edu/~eclab/projects/ecj/`

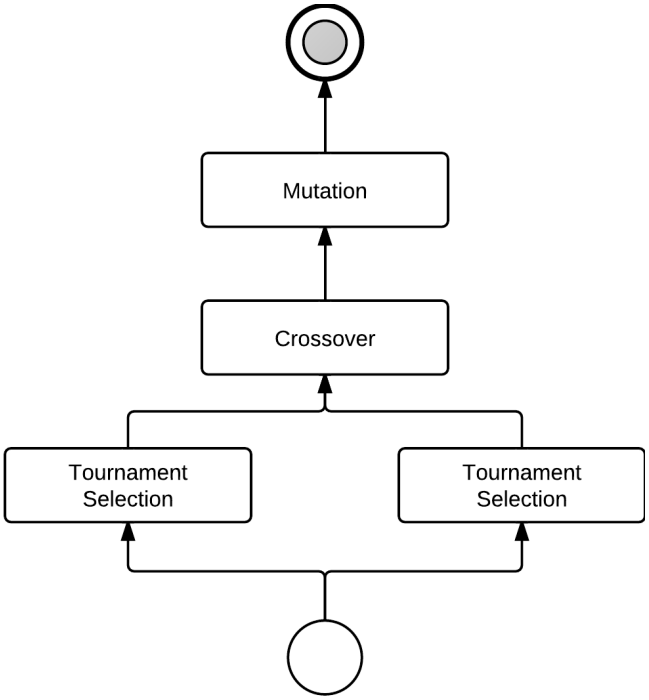

Figure 3: Genetic algorithm's breeding pipeline

Number of generations	100
Number of individuals	40
Crossover probability	0.75
Mutation probability	0.70
Replacement policy	1-elitism

Table 1: Genetic algorithm's parameters

erations. It uses tournament selection (i.e. the algorithm selects two individuals and selects the one with the higher fitness) as the selection method, and crossover and mutation as breeding operators (figure 3 shows a detailed view of the breeding pipeline). The crossover method selected for our algorithm performs a line recombination: The two individuals are treated as points in space. A straight line is drawn through both points, and two children are created along this line. If the individuals are $\overrightarrow{x}$ and $\overrightarrow{y}$, we draw two random values α and β, each between $-p$ and $1+p$ inclusive. Then the two children are defined as $\alpha\overrightarrow{x} + (1-\alpha)\overrightarrow{y}$ and $\beta\overrightarrow{y} + (1-\beta)\overrightarrow{x}$ respectively (with $p = 0.75$). We have used gaussian mutation as mutation operator. It adds Gaussian noise to the current value, this way, planets may be displaced or its size may be changed. If the result is outside the bounds of minimum and maximum legal values, another Gaussian noise is tried instead, and so on, until a legal value is found (or a certain number of retries is reached).

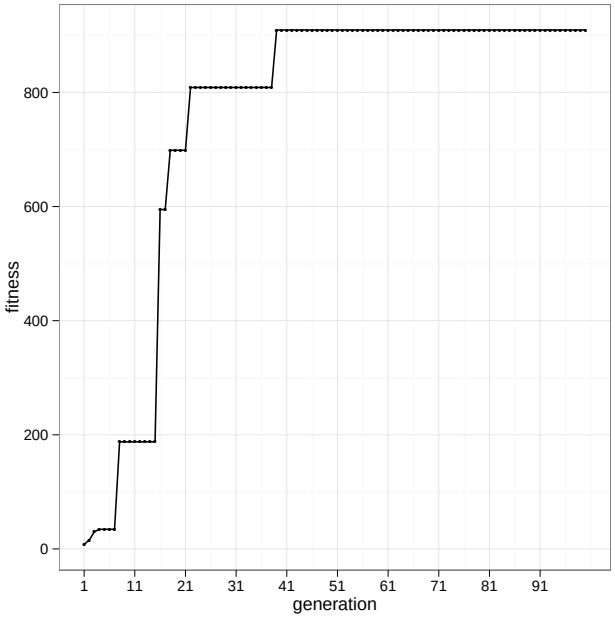

Figure 4: Evolution of the fitness.

To evaluate the fitness of every individual the algorithm makes a genotype-to-phenotype transformation, scaling the values of the individual. Positions are scaled between 10 and 50, the number of ships is rounded to an integer and scaled between 10 and 50 while the growth rate is just rounded to an integer. Once these parameters has been normalized, the algorithm writes the map to a file, using these parameters to generate the planets. Then, the algorithm runs a game that take place on the recently generated map between two players. These two players are instances of the same bot, this way the player's ability does not affect the measurement of how balanced has been this game. Once the game has finished, the algorithm gathers the total number of ships (S1 and S2 respectively) and planets (NP1 and NP2 respectively) owned by both players and compute the fitness function (1) with these values.

$$f = \frac{(NP1 * NP2)(S1 + S2)}{|S1 - S2| + 1} \quad (1)$$

We wanted to obtain balanced maps, that is, maps where a player doesn't get any advantage over the other players. This is the reason why we have the difference between the two players' number of ships on the denominator of the fitness function, because we wanted at least score difference as possible between the two players (is a balanced map). Keeping in mind the same objective, we sum these number of ships on the numerator to promote maps where the players last enough to build big fleets. This sum is multiplied by another multiplication (number of planets of both players) to penalize those maps where a player wins over another or both players remain static until the end of turns.

RESULTS

We have run this algorithm several times, and we have obtained many fully playable and balanced maps. After making these experiments we have noticed that the evolution of the fitness is not constantly growing, that is, sometimes the fitness remains unchanged during a certain amount of generations. This fitness doesn't change because this genetic algorithm has elitism as the selection method (the best individual survives and is included in the next population). Another observation is that the planets of the generated maps are much separated from each other. Maps of this kind should have a high fitness value because it takes a long time (number of turns) to reach the enemy, so the number of ships for each player increases without battles decreasing it. These neutral planets have different sizes and these sizes don't appear to follow any trend or be restricted to any range (besides the value range for the parameters of the genetic algorithm's individuals). Moreover, the position of these neutral planets are different in every map and they don't follow any trend as well.

CONCLUSION AND FUTURE WORK

In this paper we have introduced a simple procedural map generator for a RTS game that is capable of generating balanced maps for two player games in an acceptable execution time. An example of a map generated by this algorithm is shown in figure 5. Despite this algorithm generates fully playable maps, there are several improvements that could be made to this generator. For example, the generator uses a simple genetic algorithm which can be tuned more exhaustively to obtain a better performance (changing the breeding

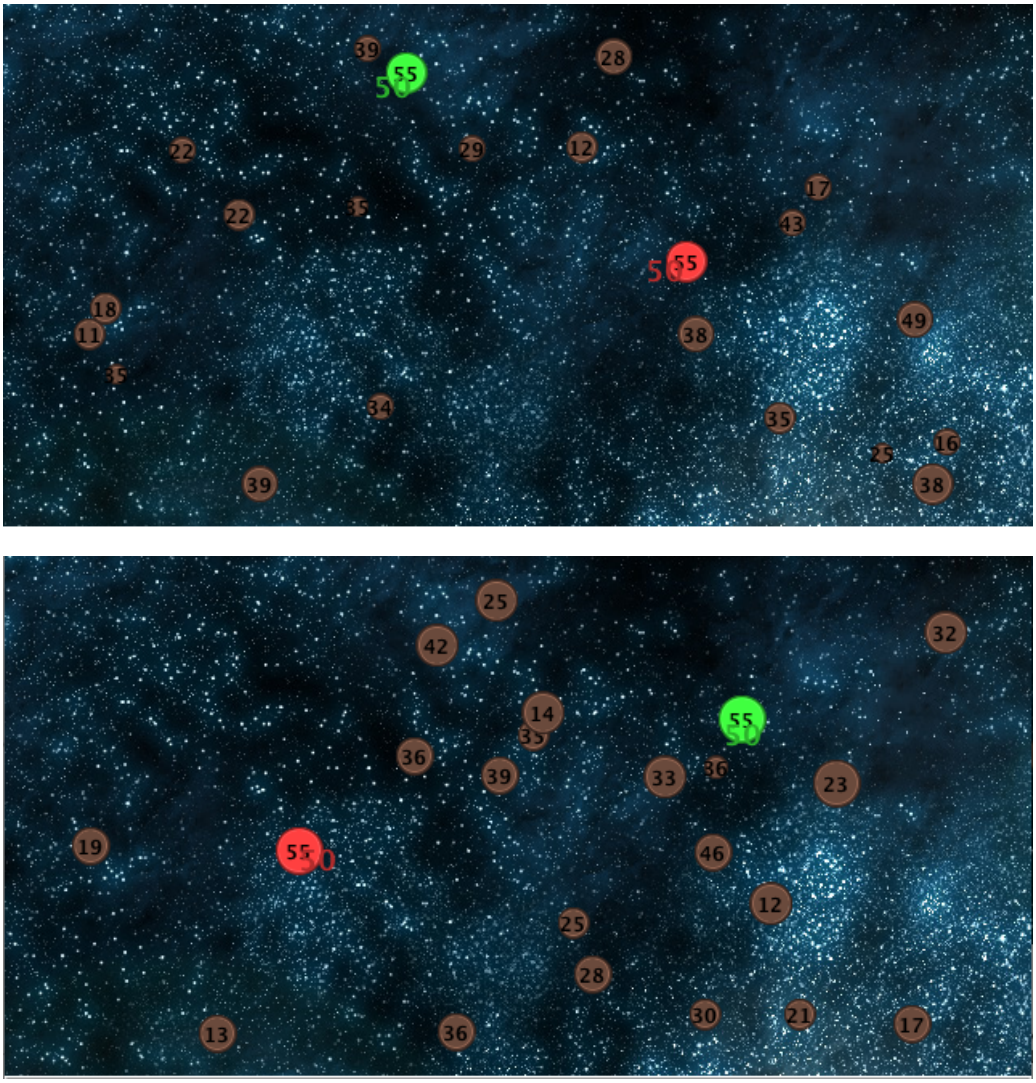

Figure 5: Balanced maps for Planet Wars.

pipeline or/and researching more optimal breeding operators). The maps generated by this algorithm are not symmetrical and some planets should be overlapped, so the map generation function could be improved avoiding overlapped planets and forcing these to be symmetrical. When evolving the maps, the fitness function is obtained from only one game (execution) with the same two players. A better fitness function should obtain its data from several games with different players playing each of these games. This way the map would not been balanced only for a kind of player, but a group of them. Although this is a simple map generator for a simple RTS game, it can be easily scaled to work with more complex games and situations.

ACKNOWLEDGEMENTS

This work is partially supported by Spanish MICINN under project ANYSELF (TIN2011-28627-C04-01), and by Junta de Andalucía under project P10-TIC-6083 (DNEMESIS).

REFERENCES

Blizzard Entertainment 1998, *Starcraft*, Blizzard Entertainment.

Frade, M., de Vega, F. & Cotta, C. 2010*a*, Evolution of artificial terrains for video games based on accessibility, *in* C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekrt, A. Esparcia-Alcazar, C.-K. Goh, J. Merelo, F. Neri, M. Preu, J. Togelius & G. Yannakakis, eds, 'Applications of Evolutionary Computation', Vol. 6024 of *Lecture Notes in Com-*

puter Science, Springer Berlin / Heidelberg, pp. 90–99.

Frade, M., de Vega, F. F. & Cotta, C. 2008, Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience, *in* M. Giacobini et al., eds, 'Applications of Evolutionary Computing', Vol. 4974 of *Lecture Notes in Computer Science*, Springer, pp. 485–490.

Frade, M., de Vega, F. F. & Cotta, C. 2009, 'Breeding terrains with genetic terrain programming: The evolution of terrain generators', *International Journal of Computer Games Technology* **2009**.

Frade, M., de Vega, F. F. & Cotta, C. 2010*b*, Evolution of artificial terrains for video games based on obstacles edge length, *in* 'IEEE Congress on Evolutionary Computation', IEEE, pp. 1–8.

Gearbox Software 2009, *Borderlands*, 2K Games.

Mahlmann, T., Togelius, J. & Yannakakis, G. N. 2012, Spicing up map generation, *in* C. D. Chio et al., eds, 'EvoApplications', Vol. 7248 of *Lecture Notes in Computer Science*, Springer, pp. 224–233.

Maxis 2008, *Spore*, Electronic Arts.

Mojang 2011, *Minecraft*, Mojang.

Togelius, J., De Nardi, R. & Lucas, S. 2007, Towards automatic personalised content creation for racing games, *in* 'Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on', pp. 252–259.

Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelback, J. & Yannakakis, G. 2010, Multiobjective exploration of the starcraft map space, *in* 'Computational Intelligence and Games (CIG), 2010 IEEE Symposium on', pp. 265 –272.

Togelius, J., Preuss, M. & Yannakakis, G. N. 2010, Towards multiobjective procedural map generation, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', pp. 3:1–3:8.

Togelius, J., Yannakakis, G. N., Stanley, K. O. & Browne, C. 2011, 'Search-based procedural content generation: A taxonomy and survey', *IEEE Transactions on Computational Intelligence and AI in Games* **3**(3), 172–186.

White, D. 2012, 'Software review: the ecj toolkit', *Genetic Programming and Evolvable Machines* **13**, 65–67. 10.1007/s10710-011-9148-z.