A procedural balanced map generator with self-adaptive complexity for the real-time strategy game Planet Wars

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

Department "Lenguajes y Ciencias de la Computación", ETSI Informática,
University of Málaga, Campus de Teatinos, 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 gameplay spaces. This methodology brings many advantages to game developers, such as reduced memory consumption. This works presents a procedural balanced map generator for a real-time strategy game: *Planet Wars*. This generator uses an evolutionary strategy for generating and evolving maps and a tournament system for evaluating the quality of these maps in terms of their balance. We have run several experiments obtaining a set of playable and balanced maps

1 Introduction

Procedural content generation (PCG) refers to creating game content automatically, through algorithmic means. This content refers to all aspects of the game that affect gameplay other than non-player characters (NPCs), 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.

Due to the benefits detailed previously, procedural content generation has been used in many well-known videogames. *Borderlands* [10] 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* [15], a sandbox-building game with an infinite map which is expanded dynamically. *Spore* [14] 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 [19] 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 [21] 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 [20]. In this case, the maps are generated for the game *Starcraft* [2]. Frade et al. have introduced the idea of terrain programming, namely the use of genetic programming to evolve playing maps for videogames, using either subjective human-based feedback [7], [8] or automated quality measures such as accessibility [6] or edge-length [9]. In [13] 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.

Real-time strategy (RTS) games are a genre of videogames which require managing different kind of units and resources in real-time. In a RTS game the participants position and maneuver units and structures under their control to secure areas of the map and/or destroy their opponents' assets. In a typical RTS, it is possible to create additional units and structures during the course of a game, but this is generally limited by the number of accumulated resources. These resources are gathered by controlling special points on the map and/or possessing certain types of units and structures devoted to this purpose. The typical game of the RTS genre features resource gathering, base building, in-game technological development and indirect control of units. They are usually played by two or more players (human or not). These players have to deal with incomplete information during the game (the map is covered by fog of war, the technology developed by a player is unknown by every other player, ...). These features make RTS games a great tool for computational intelligence research, since a RTS game player needs to master many challenging problems such as resource allocation [3,11], strategy planning [1,5,16] and opponent's strategy prediction [4,18]. In addition, procedural content generation can be used to create maps, units and technologies for RTS games. Traditionally, academic game artificial intelligence (AI) was mainly linked to non player character (NPC) behavior and pathfinding. However, there are new research areas that have recently provided innovative solutions for a number of game development challenges, like player experience modeling (PEM), procedural content generation (PCG) and large scale game data mining [23].

This paper introduces a map generation method for a RTS game that can be categorized (using the taxonomy proposed in [22]) as an off-line method that generates necessary content, using random seeds and deterministic generation and following a generate-and-test schema. This method generates balanced maps, i.e. maps where players do not have any advantage over their opponents regardless of their ability or strategy type.

2 Game Description

Planet Wars is a real-time strategy (RTS) game based on *Galcon* and used in the *Google AI Challenge 2010*. The game is set in outer space and its objective is to take over all the planets on the map or eliminate all of your opponents ships. A game of Planet Wars takes place on a map that contains several planets with some number of ships on it. Each planet may have a different number of ships. The planets may belong to some player or may be neutral. The game has a certain maximum number of turns and it may end earlier if one of the players loses all his ships, and in this case the player that has ships remaining wins instantly. If both players have the same number of ships when the game ends, it is considered 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 those 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.

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. A planet can be neutral or owned by some player. 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. It is given as an integer and it also represents the size (i.e. radius) of the planet. 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.

3 A Procedural Balanced Map Generator

A map is balanced if players do not have any advantage over their opponents regardless of their ability or strategy type. Due to this feature, this kind of maps are important for the evaluation of human or artificial players, since they do not boost the performance of any player. In order to create balanced maps, we have designed a procedural map generator that is composed of an evolutionary strategy and a tournament system. The evolutionary strategy is responsible for

generating new random maps and evolving them, while the tournament system evaluates the quality of the generated maps based on the results obtained from several matches between non-player characters.

3.1 Evolutionary strategy

As mentioned before, the evolutionary strategy (ES) is devoted to generate maps with an arbitrary number of neutral planets ranged between 15 and 30 following the rules of the game. These maps are the individuals of the ES and they are represented by a variable-length vector of planets. As described on the previous section, every planet has five properties: x-position, y-position, owner, growth rate and number of ships. We have fixed planet's holders so that players own the first and second planet of every map while the rest of the planets are neutral, so individuals' genes are groups of 4 parameters. In addition to these parameters the algorithm needs 4 additional parameters since this is a self-adaptive evolutionary strategy so the parameters of the mutation operator evolve along with the planets' parameters.

Regarding these planets' parameters, two of them have real values (x and y position) while the other two (growth rate and number of ships) have integer values. In addition to this, x and y positions range between 0 and 15, while the growth rate and the number of ships fluctuate between 1 and 5, and 100.

Due to the types of the parameters (real and integer), the evolutionary strategy uses an hybrid mutation operator that uses different mutation methods for real and integer parameters. The operator mutates x and y coordinates following a Gaussian mutation scheme with self-adaptive step sizes. The problem with applying Gaussian mutation to integer values is that this kind of mutation generates real-value perturbations which are rounded to an integer perturbation. To prevent this, this mutation operator uses a method [12,17] that generates suitable integer mutations for the growth rate and number of ships. This method is similar to the self-adaptive mutation of real values, with a set of step-size parameters controlling the strength of the mutation, but using the difference of two geometrically distributed random variables to generate the perturbation instead of the normal distributed random variables used by the real values method.

In the case of real-valued parameters $\langle x_1, ..., x_n \rangle$ they are extended with n step sizes, one for each parameter, resulting in $\langle x_1, ..., x_n, \sigma_1, ..., \sigma_n \rangle$. The mutation mechanism is specified as follows:

$$\sigma_i' = \sigma_i \cdot e^{\tau' \cdot N(0,1) + \tau \cdot N_i(0,1)}$$
$$x_i' = x_i + \sigma_i \cdot N_i(0,1)$$

where $\tau' \propto 1/\sqrt{2n}$, and $\tau \propto 1/\sqrt{2\sqrt{n}}$. A boundary rule is applied to step sizes to prevent standard deviations very close to zero: $\sigma_i' < \epsilon_0 \Rightarrow \sigma_i' = \epsilon_0$ (in this algorithm, σ_0 represents 1% of the parameter's range).

Regarding integer-valued parameters $\langle z_1, ..., z_m \rangle$ they are extended in a similar way than real-valued parameters, resulting in $\langle z_1, ..., z_m, \varsigma_1, ..., \varsigma_m \rangle$. The mu-

tation mechanism is specified as follows:

$$\varsigma_i' = \max(1, \varsigma_i \cdot e^{\tau \cdot N(0,1) + \tau' \cdot N(0,1)})$$
$$\psi_i = 1 - (\varsigma_i'/m) \left(1 + \sqrt{1 + \left(\frac{\varsigma_i'}{m}\right)^2}\right)^{-1}$$
$$z_i' = z_i + \left\lfloor \frac{ln(1 - U(0,1))}{ln(1 - \psi_i)} \right\rfloor - \left\lfloor \frac{ln(1 - U(0,1))}{ln(1 - \psi_i)} \right\rfloor$$

where $\tau = 1/\sqrt{2m}$ and $\tau' = 1/\sqrt{2\sqrt{m}}$. As described before, the main difference between the two methods is the distribution used to generate the perturbation.

Continuing with operators, this evolutionary strategy uses a "cut and splice" operator that recombines two individuals by swapping cut pieces with different sizes (this way, generated maps have different numbers of planets). We have chosen this operator due to the arbitrary length of the individuals. Table 1 summarizes the algorithm's parameters.

Representation	Vector of planets
Recombination	Cut and slice
Mutation	Gaussian perturbation (real) and geometric difference (integers)
Parent selection	Binary tournament
Survivor selection	$(\mu + \lambda)$ with $\mu = 10$ and $\lambda = 100$
Speciality	Self-adaption of mutation step sizes and genome length

Table 1. Algorithm's parameters

To evaluate the quality of every individual the algorithm runs a tournament that takes place on the generated map between several players. Once the tournament has finished, the algorithm gathers the individual's fitness from the result of the tournament. Equation (3) defines the fitness, with N_m being the number of matches played during the tournament, t_i being the number of turns of match i, K_i being the added up percentage of occupied planets by both players at the end of the game, $P_{ij}^{(1)}$, $P_{ij}^{(2)}$ being the percentage of owned planets by player 1 and player 2 respectively, in match i and turn j and $S_{ij}^{(1)}$, $S_{ij}^{(2)}$ being the percentage of the total ships owned by player 1 and player 2 respectively in match i and turn j.

$$\bar{P}_i = \frac{\sum_{j=1}^{t_i} \left| P_{ij}^{(1)} - P_{ij}^{(2)} \right|}{t_i} \tag{1}$$
$$\bar{S}_i = \frac{\sum_{j=1}^{t_i} \left| S_{ij}^{(1)} - S_{ij}^{(2)} \right|}{t_i} \tag{2}$$

$$fitness = \left(\frac{1}{N_m} \sum_{i=1}^{N_m} \frac{K_i \cdot t_i}{\bar{P}_i + \bar{S}_i + 1} \right)^2 \tag{3}$$

Fitness function (3) promotes balanced maps through its components: $\bar{P}_i$ and $\bar{S}_i$ promotes maps where players have similar number of planets and ships (it sums up 1 to avoid dividing by zero), while t_i promotes long games because it means that there have not been a winner or the winner is determined nearly at the end of the game. Finally, K_i promotes maps where there have been high activity, i.e. players have conquered many planets.

3.2 Tournament System

The tournament system is the component devoted to evaluate the quality of the generated maps. This component runs a set of *Planet Wars* games between an arbitrary number of non-player characters (NPC). Every NPC plays at least a game against each other, although this parameter is customizable. The tournament system evaluates every game analyzing the logs generated by a Java console-style tool, which was developed by *Google* for the *Google AI Challenge 2010*. The evolutionary strategy provides the maps to the tournament system, which evaluates the map and returns this evaluation to the former.

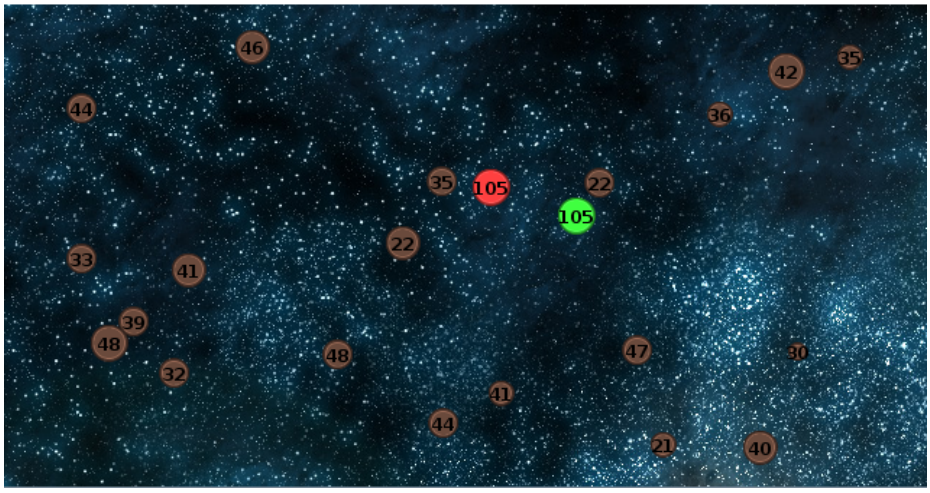

Fig. 1. An example of a balanced procedural generated map.

4 Experiments and Results

We have run two experiments (10 executions each) with different parameters, obtaining a set of playable and balanced maps (one of these maps is shown in

Figure 1). The first experiment uses an evolutionary strategy with the parameters described before (see Table 1), using a self-adapting strategy for mutation steps and genome length (i.e. number of planets in the map), while the second experiment uses the same parameters except for the fixed genome length (23 planets in every map since the number of planets ranges between 15 and 30). We have evaluated the quality of the maps using the tournament system with three NPCs who were participants of the *Google AI Challenge 2010* (*Manwe*[1], *Flagscapper's bot*[2] and *fglider's bot*[3]) , all of them ranked in the top 100 and having their source code available (there were over 4600 participants). The maximum number of turns per game has been limited to 400 turns. We have observed that the planets of many generated maps are much separated from each other. Maps of this kind should be considered as balanced maps because it takes a long time (number of turns) to reach the enemy and fight him, so players can conquest new planets without troubles and their fleet grows with a similar rate —the fitness of this kind of maps will be high because of the low difference between the number of owned planets and ships.

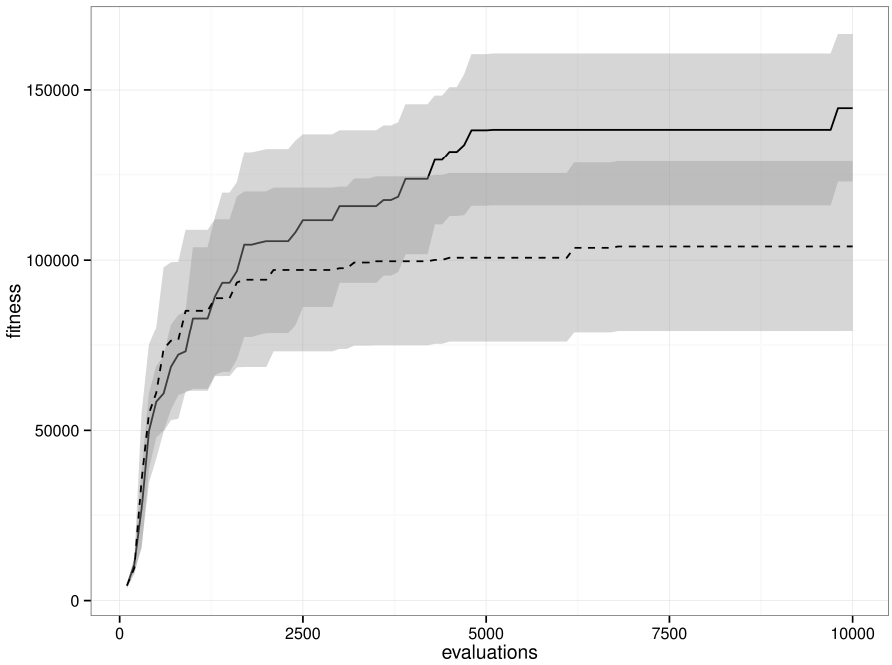

Fig. 2. Evolution of the averaged fitness.

[1] `https://github.com/Manwe56/Manwe56-ai-contest-planet-wars`
[2] `http://flagcapper.com/?c1`
[3] `http://planetwars.aichallenge.org/profile.php?user_id=8490`

Figure 2 shows the evolution of the averaged fitness for the two experiments (solid line for self-adaption of the genome size and mutation steps and dashed line for self-adaption of mutation steps only). Grey areas show the standard mean error of the averaged fitness values. As we can see in the figure, both experiments have a similar behavior over the first evaluations but the self-adaptive algorithm (experiment 1) gets a better fitness over the subsequent evaluations. Figure 3 shows the evolution of the averaged number of planets in the best map (i.e. the individual with the highest fitness). As we can observe in the figure, after some evaluations, this number converges to the value 17, so we should think that maps with 17 planets are more balanced than other maps with a higher number of planets.

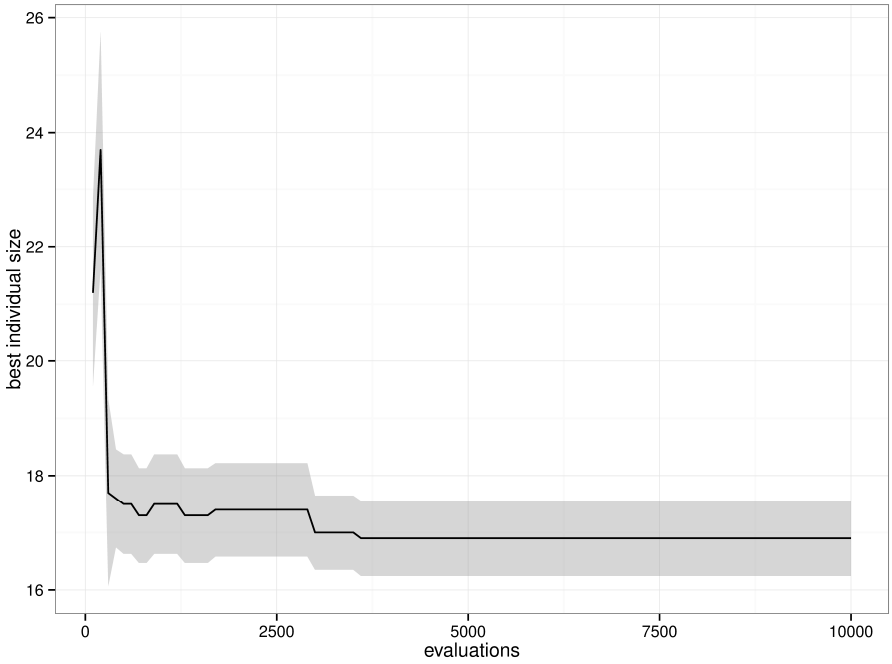

Fig. 3. Evolution of the averaged number of planets in the best map.

5 Conclusion and Future Work

In this paper we have introduced a procedural map generator for a RTS game that is capable of generating balanced maps for a real-time strategy game: *Planet Wars*. These maps do not give any advantage to the players regardless of their ability or strategy type. This generator turns *Planet Wars* into an endless game

and makes the game more interesting to weak players (since they do not lose with ease), raising the competitiveness of the stronger player with harder challenges.

Despite this algorithm generates fully playable maps, there are some improvements that could be made to this generator. The evolutionary strategy uses only a mutation operator, so it could be interesting to improve the breeding pipeline, adding additional or improved mutation and recombination operators. Moreover, maps generated by this algorithm are not symmetrical and some planets should be overlapped, although the evolutionary strategy avoids overlapped planets since this is an advantage to the player who has this overlapped planets nearer. In addition to this, it is possible to obtain other characteristics of the maps that make them more balanced, such as the averaged distance between the planets, players' initial positions or the distribution of the planets over the map.

In the near future, we are going to introduce interactivity and pro-activity to this procedural map generator, in order to improve its performance and the quality of generated maps.

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

1. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to win: Case-based plan selection in a real-time strategy game. In: Muñoz-Avila, H., Ricci, F. (eds.) International Conference on Case-Based Reasoning. Lecture Notes in Computer Science, vol. 3620, pp. 5–20. Springer (2005)
2. Blizzard Entertainment: Starcraft. Blizzard Entertainment (1998)
3. Chan, H., Fern, A., Ray, S., Wilson, N., Ventura, C.: Online planning for resource production in real-time strategy games. In: Boddy, M.S., et al. (eds.) International Conference on Automated Planning and Scheduling. pp. 65–72. The AAAI Press (2007)
4. Cheng, D., Thawonmas, R.: Case-based plan recognition for real-time strategy games. In: El-Rhalibi, A., van Welden, D. (eds.) GameOn Conference. pp. 36–40. EUROSIS (2004)
5. Chung, M., Buro, M., Schaeffer, J.: Monte Carlo Planning in RTS Games. In: IEEE Symposium on Computational Intelligence and Games. IEEE (2005)
6. Frade, M., de Vega, F., Cotta, C.: Evolution of artificial terrains for video games based on accessibility. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekrt, A., Esparcia-Alcazar, A., Goh, C.K., Merelo, J., Neri, F., Preu, M., Togelius, J., Yannakakis, G. (eds.) Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 6024, pp. 90–99. Springer Berlin / Heidelberg (2010)
7. Frade, M., de Vega, F.F., Cotta, C.: Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience. In: Giacobini, M., et al. (eds.) Applications of Evolutionary Computing. Lecture Notes in Computer Science, vol. 4974, pp. 485–490. Springer (2008)

8. Frade, M., de Vega, F.F., Cotta, C.: Breeding terrains with genetic terrain programming: The evolution of terrain generators. International Journal of Computer Games Technology 2009 (2009)
9. Frade, M., de Vega, F.F., Cotta, C.: Evolution of artificial terrains for video games based on obstacles edge length. In: IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2010)
10. Gearbox Software: Borderlands. 2K Games (2009)
11. Kovarsky, A., Buro, M.: A First Look at Build-Order Optimization in Real-Time Strategy Games. In: Wolf, L., Magnor, M. (eds.) GameOn Conference. pp. 18–22. EUROSIS (2006)
12. Li, R.: Mixed-integer evolution strategies for parameter optimization and their applications to medical image analysis. Ph.D. thesis (2009)
13. Mahlmann, T., Togelius, J., Yannakakis, G.N.: Spicing up map generation. In: Chio, C.D., et al. (eds.) EvoApplications. Lecture Notes in Computer Science, vol. 7248, pp. 224–233. Springer (2012)
14. Maxis: Spore. Electronic Arts (2008)
15. Mojang: Minecraft. Mojang (2011)
16. Ng, P.H.F., Li, Y.J., Shiu, S.C.K.: Unit formation planning in RTS game by using potential field and fuzzy integral. In: Fuzzy Systems. pp. 178–184. IEEE (2011)
17. Rudolph, G.: An evolutionary algorithm for integer programming. In: Davidor, Y., Schwefel, H.P., Mnner, R. (eds.) Parallel Problem Solving from Nature PPSN III, Lecture Notes in Computer Science, vol. 866, pp. 139–148. Springer Berlin Heidelberg (1994)
18. Synnaeve, G., Bessiere, P.: A bayesian model for opening prediction in RTS games with application to StarCraft. In: Computational Intelligence and Games. pp. 281 –288. IEEE (2011)
19. Togelius, J., De Nardi, R., Lucas, S.: Towards automatic personalised content creation for racing games. In: Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on. pp. 252–259 (2007)
20. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelback, J., Yannakakis, G.: Multiobjective exploration of the starcraft map space. In: Computational Intelligence and Games (CIG), 2010 IEEE Symposium on. pp. 265 –272 (2010)
21. Togelius, J., Preuss, M., Yannakakis, G.N.: Towards multiobjective procedural map generation. In: Proceedings of the 2010 Workshop on Procedural Content Generation in Games. pp. 3:1–3:8 (2010)
22. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation: A taxonomy and survey. IEEE Transactions on Computational Intelligence and AI in Games 3(3), 172–186 (2011)
23. Yannakakis, G.N.: Game ai revisited. In: Proceedings of the 9th conference on Computing Frontiers. pp. 285–292. CF '12, ACM, New York, NY, USA (2012)