# Spicing up map generation

Tobias Mahlmann, Julian Togelius and Georgios N. Yannakakis

IT University of Copenhagen, Rued Langaards Vej 7, 2300 Copenhagen, Denmark
{tmah, juto, yannakakis}@itu.dk

**Abstract.** We describe a search-based map generator for the classic real-time strategy game *Dune 2*. The generator is capable of creating playable maps in seconds, which can be used with a partial recreation of Dune 2 that has been implemented using the Strategy Game Description Language. Map genotypes are represented as low-resolution matrices, which are then converted to higher-resolution maps through a stochastic process involving cellular automata. Map phenotypes are evaluated using a set of heuristics based on the gameplay requirements of Dune 2.

## 1 Introduction

Procedural Content Generation (PCG) for Games is a field of growing interest among game developers and academic game researchers alike. It addresses the algorithmic creation of new game content. Game content normally refers to weapons, textures, levels or stories etc. and may — to distinguish PCG from other fields of research — exclude any aspect connected to agent behaviour, although generating behavioural policies might be considered PCG in some contexts. One particular approach to PCG which has gained traction in recent years is the *search-based* paradigm, where evolutionary algorithms or other stochastic optimisation algorithms are used to search spaces of game content for content artefacts that satisfy gameplay criteria [14]. In search-based PCG, two of the main concerns are how this content is represented and how it is evaluated (the fitness function). The key to effective content generation is largely to find a combination of representation and evaluation such that the search mechanism quickly zooms in on regions of interesting, suitable and diverse content.

We are addressing the problem of map generation, in particular the generation of maps for a strategy game. A "map" is here taken to mean a two-dimensional spatial structure (though maps for some other types of games might be three-dimensional) on which objects or features of some kind (e.g. trees, tanks, mountains, oil wells, bases) are placed and on which gameplay takes place. While the generation of terrains without particular reference to gameplay properties is a fairly well-studied problem [13, 5, 6, 3, 2], a smaller body of work has addressed the problem of generating maps such that the maps support the game mechanics of a particular game or game genre.

One example of the latter is the cave generator by Johnson et al. [8], which generates smooth two-dimensional cave layouts, that support the particular design needs of a two-dimensional abusive endless dungeon crawler game. This basic principle of that generator is to randomly sprinkle "rock" and "ground" on an open arena, and then use cellular automata (CA) to "smelt the rock together" in several steps, after which another heuristic ensures that rooms are connected to each other. While the resulting generator is fast enough for on-the-fly generation and generates natural-looking and adequately functional structures, the CA-based method lacks controllability and could not easily be adapted to generate maps that satisfy other functional constraints (e.g. reachability).

Another example is the search-based map generator for the real-time strategy game *StarCraft* by Togelius et al. [15]. Recognising that devising a single good evaluation function for something as complex as a strategy game map is anything but easy, the authors defined a handful of functions, mostly based on distance and path calculations, and used multi-objective evolutionary algorithms to study the interplay and partial conflict between these evaluation dimensions. While providing insight into the complex design choices for such maps, it resulted in a computationally expensive map generation process and problems with finding maps that are "good enough" in all relevant dimensions. The map representation is a combination of direct (positions of bases and resources) and indirect (a turtle-graphics-like representation for rock formations), with mixed results in terms of evolvability.

We propose a new search-based method for generating maps that draws heavily on the two very different approaches described above. Like in the StarCraft example, we use an evolutionary algorithm to search for maps and a collection of heuristics derived from an analysis of the game's mechanics to evaluate them. The embryogeny is borrowed from the cave generator. The transformation from genotype (which is evolved) to phenotype (which is evaluated) is happening through a process of sprinkling and smelting trough cellular automata. These steps will be described in some detail below. Our results show that this process effectively generates maps that look good and satisfy the specifications. The target game in this paper is *Dune 2*, which has the advantage of being in several respects simpler than StarCraft, which makes it easier to craft heuristic evaluation functions based on its mechanics, and also makes it easier to re-implement it in our own strategy game modelling framework for validating the results.

This paper is an integral part of the Strategy Games Description Language (SGDL) project at IT University of Copenhagen. SGDL is an initiative to model game mechanics of strategy games. Our previous work consisted of evolving heterogeneous unit sets [10], different approximations of game play quality [9], and general purpose agents for strategy games [12]. The re-creation of Dune 2 as a turn-based strategy game is a continuation of this research. An example map, created by the generator described in this paper, loaded into the SGDL game engine can be seen in Figure 2.

## 2 Background

Dune 2 (Westwood 1992) is one of the earliest examples of real-time strategy games, and came to strongly influence this nascent genre. The game is loosely based on Frank Herbert's Dune [7] but introduces new plots and acting parties.

The player takes the role of a commander of one of three dynasties competing in the production of “spice”, a substance that can only be gathered on the desert planet “Arrakis”, also known as “Dune”. In the dune universe, spice is required for inter-stellar travel, making it one of the most valuable substance in the universe. Dune 2 simplifies this relation slightly, treating spice as a resource which can be used to build new units and buildings. The only way to gain spice is sending harvester units to the sand parts of the map, where the spice is located. Apart from opposing parties that try to harvest the same fields, the sand parts are also habited by the native animals of the planets: the sandworms. Menace and important resource alike, these non-controllable units are involved in the generation of new spice on the map, but also occasionally swallowing units of the player - or his enemies if he uses the sand as a tactical element.

The main objective of the player on each map is to harvest spice and use the gathered resources to build new buildings and produce military units to ultimately destroying one or two enemies’ bases. As mentioned, compared to modern real-time strategy games the game is rather simple: there is only one resource, two terrain types and no goals beside eliminating the enemies’ forces. The two terrain types are “rocky terrain” and “sand”, and both can be passed by all units. Two game mechanics involve the terrain types: buildings can only be constructed on rocky terrain, and spice and sandworms can only exist on sand. For completeness it should be mentioned that the game also contains cliffs that are only passable by infantry, but those have negligible effect on gameplay. Although the game does not contain any mechanics to model research, buildings and units are ordered in tiers. As the single player campaign progresses, the game simply unlocks additional tiers as the story progresses. This removes the necessity to model additional mechanics. An exemplary screenshot of the original game can be seen in Figure 1.

## 3 Map generator

The map generator consists of two parts: the genotype-to-phenotype mapping and the search-based framework that evolves the maps. The genotypes are vectors of real numbers, which serve as inputs for a process that converts them to phenotypes, i.e. complete maps, before they are evaluated. The genotype-to-phenotype mapping can also be seen as, and used as, a (constructive) map generator in its own right. (The relationship between content generators at different levels, where one content generator can be used as a component of another, is discussed further in [14].)

*The genotype-to-phenotype mapping* is a constructive algorithm that takes an input as described in the following and produces an output matrix $o$. Based on tile types of the original Dune 2, the elements of $o$ can assume the value $0 =$ SAND, $1 =$ ROCK, and $2 =$ SPICE. The matrix $o$ is then later interpreted by a game engine into an actual game map. Our current implementation contains only an SGDL backend, but using an open source remake of the game and its

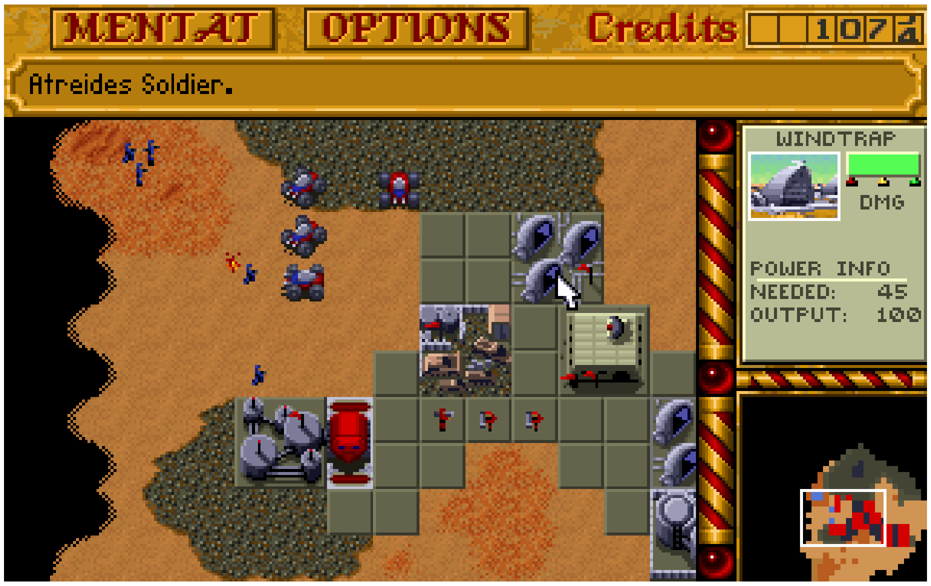


**Fig. 1.** Screenshot from the original Dune 2 showing the player's base with several buildings, units, and two spice fields in direct proximity.

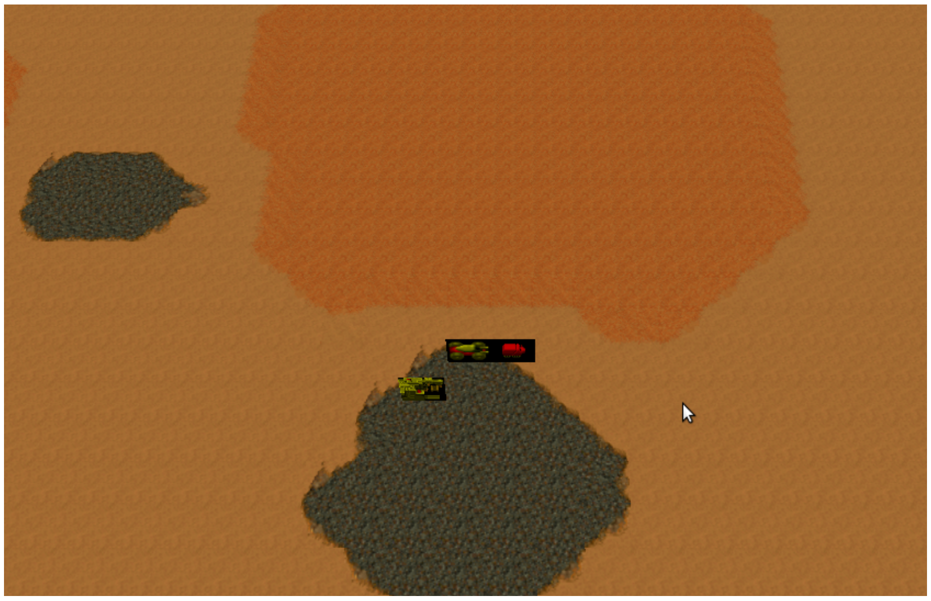


**Fig. 2.** A Dune 2 map loaded into the SGDL Game Engine. Terrain and unit textures are taken from the original asset set, but actors are, due to the lack of 3D models, placed as billboards into the game world.

tools (e.g. Dune II The Maker [1]) should make creating maps for the original Dune 2 easy.

The input vector is structured as followed ($mapSize$ refers to the map's edge length):

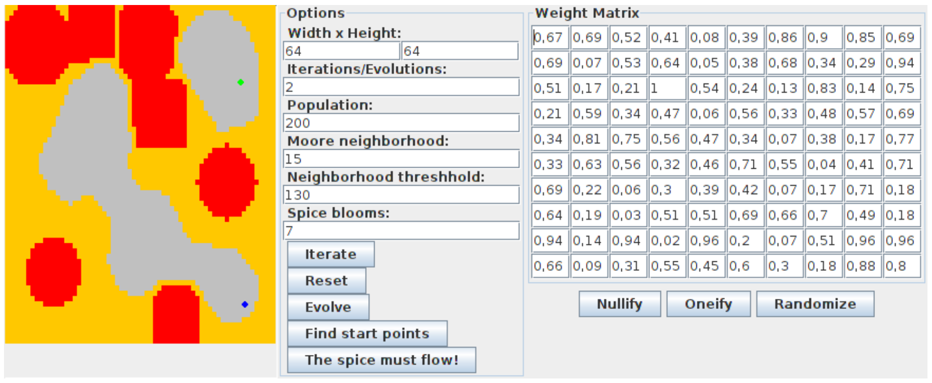
- $n$ the size of the Moore-neighbourhood $[1, \frac{mapSize}{2}]$
- $n_t$ the Moore-neighbourhood threshold $[2, mapSize]$
- $i$ the number of iterations for the CA $[1, 5]$
- $w_{00}..w_{99}$ members the weight matrix $w$ for the initial noise map $[0, 1]$
- $s$ the number of spice blooms to be placed on the map $[1, 10]$

The generator starts with creating the initial map based on the values $w$. The $10x10$ matrix is scaled to the actual map size and used as an overlay to determine the probability of a map tile starting as rock or sand. For each iteration $i_n$ a CA is invoked for each map tile to determine its new type. If the number of rock tiles in the $n$-Moore-Neighbourhood is greater or equal than $n_t$ the tile is set to *Rock* in the next iteration.

The next step is the determination of the start zones, where the players' first building will be placed. We always use the largest rock area available as the starting zones. The selection is done by invoking a 2D variant of Kadane's algorithm [4] on $o$ to find the largest sub-matrix containing ones. To prevent players from spawning too close to each other, we invoke Kadane's algorithm on a sub-matrix of $o$ that only represents the $i$ top rows of $o$ for one player, and only the $i$ bottom rows for the other player. We let $i$ run from 8 to 2 until suitable positions for both players are found. This operation ensures that one player starts in the upper half of the map and one in the lower. It also restricts us to maps that are played vertically, but this could be changed very easily. At this step we don't assert that the start positions are valid in terms of gameplay. Broken maps are eliminated through the fitness functions and the selection mechanism.

The last step is the placement of the spice blooms and filling their surrounding areas. Since Kadane's algorithm finds sub-matrices of ones, we simply clone $o$ and negate its elements with $o_{nm} = 1 - o_{nm}$; whereas $o_{nm}$ is the $m$-th member of the $n$-th row of $o$. We use the computed coordinates to fill the corresponding elements in $o$ with spice. In order to make the fields look a bit more organic, we use a simple quadratic falloff function: a tile is marked as spice if its distance $d$ from the center of the spice field (the bloom) fulfils the condition $\frac{1}{d^2} \geq t$. Where $t$ is the width of the spice field multiplied by 0.001. We created a simple frontend application to test the generator. A screenshot with a basic description can be seen in Figure 3.

*The genetic algorithm* optimises a genome in the shape of a vector of real-numbers, using a fitness function we created. Since a desert is very flat, there exists almost no impassable terrain, hence choke points (as introduced in [15]) is not a useful fitness measure. The challenge of choke points was instead replaced by the assumption that passing sand terrain can be rather dangerous due to sandworms. Furthermore, it should be ensured that both players have an equally sized starting (rock) zone and the distance to the nearest spice bloom should be



Width x Height: 64 64
Iterations/Evolutions: 2
Population: 200
Moore neighborhood: 15
Neighborhood threshhold: 130
Spice blooms: 7

Weight Matrix

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0,67 | 0,69 | 0,52 | 0,41 | 0,08 | 0,39 | 0,86 | 0,9 | 0,85 | 0,69 |
| 0,69 | 0,07 | 0,53 | 0,64 | 0,05 | 0,38 | 0,68 | 0,34 | 0,29 | 0,94 |
| 0,51 | 0,17 | 0,21 | 1 | 0,54 | 0,24 | 0,13 | 0,83 | 0,14 | 0,75 |
| 0,21 | 0,59 | 0,34 | 0,47 | 0,06 | 0,56 | 0,33 | 0,48 | 0,57 | 0,69 |
| 0,34 | 0,81 | 0,75 | 0,56 | 0,47 | 0,34 | 0,07 | 0,38 | 0,17 | 0,77 |
| 0,33 | 0,63 | 0,56 | 0,32 | 0,46 | 0,71 | 0,55 | 0,04 | 0,41 | 0,71 |
| 0,69 | 0,22 | 0,06 | 0,3 | 0,39 | 0,42 | 0,07 | 0,17 | 0,71 | 0,18 |
| 0,64 | 0,19 | 0,03 | 0,51 | 0,51 | 0,69 | 0,66 | 0,7 | 0,49 | 0,18 |
| 0,94 | 0,14 | 0,94 | 0,02 | 0,96 | 0,2 | 0,07 | 0,51 | 0,96 | 0,96 |
| 0,66 | 0,09 | 0,31 | 0,55 | 0,45 | 0,6 | 0,3 | 0,18 | 0,88 | 0,8 |

**Fig. 3.** Screenshot of the generator application. The right pane lets the user input a seed matrix directly, or observe the result of the evolution. The middle pane can be used to either invoke the generator directly ("Iterate") or start the non-interactive evolution ("Evolve"). The other buttons allow the user to go through the map generation step-by-step. The left pane shows a preview of the last map generated: yellow = sand, gray = rock, red = spice. The blue and green dot symbolise the start positions.

equal. All values were normalised to $[0, 1]$. To summarise, the following features were part of the fitness function:

- the overall percentage of sand in the map $s$
- the euclidean distance between the two starting points $d_{AB}$
- the difference of the starting zones' sizes $\Delta_{AB}$ (to minimise)
- the difference of the distance from each starting position to the nearest spice bloom $\Delta d_s$ (to minimise)

Apart from these criteria a map was rejected with a fitness of 0 if one of the following conditions was met:

- There was a direct path (using $A^*$) between both starting positions, only traversing rock tiles. (Condition $c_1$)
- One or both start positions' size was smaller than a neighbourhood of eight. (Condition $c_2$)

The resulting fitness function was:

$$f_{map} = \begin{cases} 0 & \text{if } c_0 \vee c_1, \\ \frac{s + d_{AB} + (1 - \Delta_{AB}) + (1 - \Delta d_s)}{3} & \text{else} \end{cases}$$

In other words: the average of the components if the map passed the criteria, 0 otherwise.

We ran the genetic algorithm over 150 generations with a population size of 200. Each generation took between three and ten seconds on a modern 3.2GHz desktop PC to compute. The genetic algorithm was an off-the-shelf implementation (using the JGAP library) [11] using a uniform random distribution for the genome creation and fitness driven selection probability (40% of the top scoring genomes preserved each generation).

## 4 Results

We present the result of an example run of the GA in Figure 4. The graph shows the average fitness value for each component and the overall fitness. The increasing rock coverage slightly influences the start zone size differences, as there is less rocky terrain in the map and therefore chances are higher that it is unequally distributed. There is a steady increase of the distance between the two starting zones, but this doesn't seem to have an impact on distance to the nearest spice bloom. The development of the overall fitness shows that the excluding case (where the fitness is set to zero if the map fails one or two conditions) has a high impact on the average overall score in the first 80 generations. In the same interval, the average component scores seem steady, although eliminated maps are not removed from the average component score calculations. This suggests that these maps might be enjoyable to play despite having a continuous path between starting zones.

Instead, we ran into an interesting problem with setting the elitism threshold too low (thus preserving too many genomes unaltered every generation): on rare occasions each genome in the start population would score as zero. The GA then converged quickly towards two pathological cases, which can be seen in Figure 5(c) and 5(d). The first one only consist of sand and one spice field, and the second map only consists of rock. While the second one might not be very interesting to play, it is actually playable, given that both players start with a sufficient amount of money to build units. The sand-only map on the other hand makes it impossible to win the game, since there is no space to build any buildings.

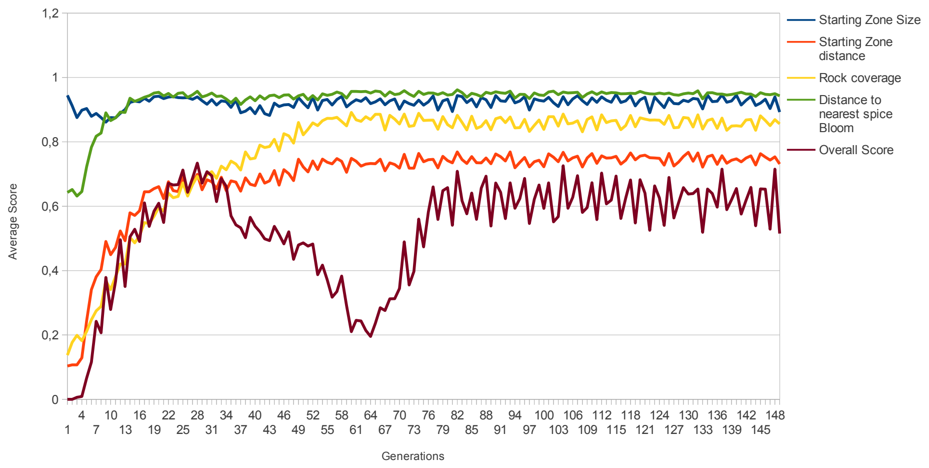
| Generation | minMax | maxMax | avgMax | stdMax |
|---|---|---|---|---|
| first | 0 | 0.82 | 0.38 | 0.39 |
| last | 0.86 | 0.92 | 0.89 | 0.02 |

**Table 1.** Aggregated results of 30 runs: the minimal maximum fitness, the maximal maximum fitness, the average maximum fitness, and the standard deviation of the maximum fitness in each the first and last generation.

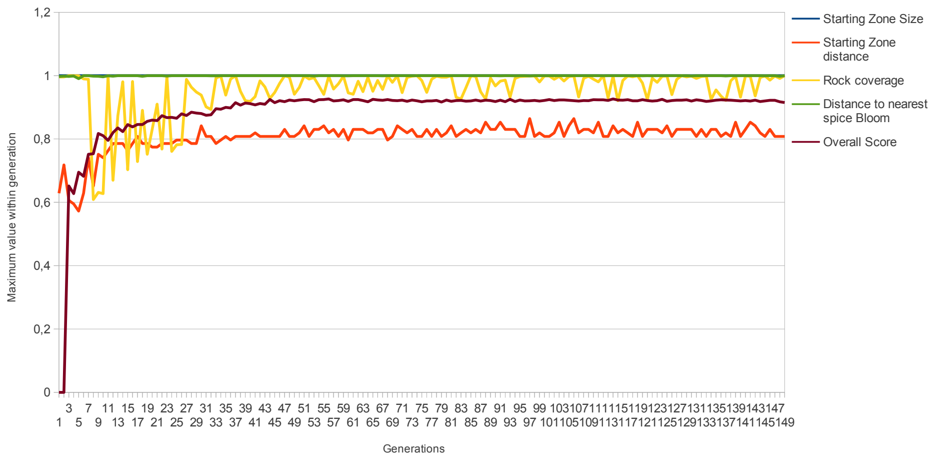
## 5 Discussion

With appropriate parameters, we were able to generate several maps that resembled the style of the original Dune 2 maps. The GA was able to adapt to our fitness function and produced "good" maps on every single run (see Table 1).

**Fig. 4.** Results from an exemplary run of the genetic algorithm.



(a) The development of the component scores and the overall fitness, displayed as the population average per generation.



(b) The overall score of the fittest genome of each generation and the maximum component value encountered in each generation. The component values are tracked individually and might come from a different individual than the fittest genome.

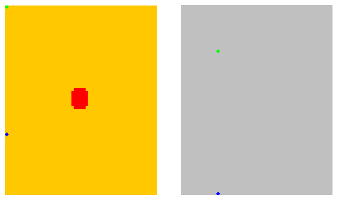

(c) Only Sand (d) Only Rock

**Fig. 5.** Two pathological, non-functional, generated maps.

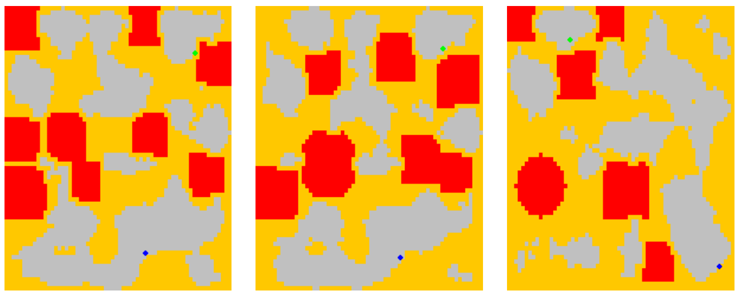

(a) 0.76 (b) 0.79 (c) 0.83

**Fig. 6.** The evolution of a map over three generations with slightly improving overall fitness.

Our fitness was based on heuristic created from expert knowledge. If this actually resembles players' preferences is clearly something that requires further examination. From an aesthetic point of view, the maps look interesting enough to not bore the player and remind them of the original Dune 2 maps, while still presenting fresh challenges.

We are currently working on modelling the complete mechanics of the original Dune 2 game in SGDL, so that both humans and AIs can play full games. We will then load the maps generated through methods described in this paper into the game and gather gameplay information and player preference reports in order to test the validity of our fitness function.

## 6 Conclusion

We have presented a fast search-based map generator that reliably generates playable and good-looking maps for Dune 2. By using a cellular automata-based genotype-to-phenotype mapping we have avoided some problems associated with other map phenotype representations, and by using a search-based mechanism with direct evaluation functions built on game mechanics we have retained controllability. We believe this method, with minor modifications, can be used to generate maps for a large variety of games.

## References

1. Dune II: The Maker, `http://d2tm.duneii.com/`
2. Ashlock, D.: Automatic generation of game elements via evolution. In: Computational Intelligence and Games (CIG), 2010 IEEE Symposium on. pp. 289 –296 (aug 2010)
3. Ashlock, D., Gent, S., Bryden, K.: Embryogenesis of artificial landscapes. In: Hingston, P.F., Barone, L.C., Michalewicz, Z. (eds.) Design by Evolution, pp. 203–221. Natural Computing Series, Springer Berlin Heidelberg (2008)
4. Bentley, J.: Programming pearls: algorithm design techniques. Commun. ACM 27, 865–873 (September 1984)
5. Doran, J., Parberry, I.: Controlled procedural terrain generation using software agents. Computational Intelligence and AI in Games, IEEE Transactions on 2(2), 111 –119 (june 2010)
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., Ekárt, A., Esparcia-Alcazar, A., Goh, C.K., Merelo, J., Neri, F., Preuß, 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. Herbert, F.: Dune. New English Library (1966)
8. Johnson, L., Yannakakis, G.N., Togelius, J.: Cellular automata for real-time generation of infinite cave levels. In: Proceedings of the 2010 Workshop on Procedural Content Generation in Games. pp. 10:1–10:4. PCGames '10, ACM, New York, NY, USA (2010)
9. Mahlmann, T., Togelius, J., Yannakakis, G.: Modelling and evaluation of complex scenarios with the strategy game description language. In: Proceedings of the Conference for Computational Intelligence (CIG) 2011. Seoul, KR (2011)
10. Mahlmann, T., Togelius, J., Yannakakis, G.: Towards procedural strategy game generation: Evolving complementary unit types. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Merelo, J., Neri, F., Preuss, M., Richter, H., Togelius, J., Yannakakis, G. (eds.) Applications of Evolutionary Computation. Lecture Notes in Computer Science, vol. 6624, pp. 93–102. Springer Berlin / Heidelberg (2011)
11. Meffert, K., Rotstan, N., Knowles, C., Sangiorgi, U.: Jgap-java genetic algorithms and genetic programming package (2008), `http://jgap.sf.net`
12. Nielsen, J.L., Jensen, B.F.: Artificial Agents for the Strategy Game Description Language. Master's thesis, ITU Copenhagen (2011)
13. Smelik, R.M., Kraker, K.J.D., Groenewegen, S.A., Tutenel, T., Bidarra, R.: A survey of procedural methods for terrain modelling. In: Proc. of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS (2009)
14. Togelius, J., Yannakakis, G., Stanley, K., Browne, C.: Search-based procedural content generation: A taxonomy and survey. Computational Intelligence and AI in Games, IEEE Transactions on 3(3), 172 –186 (sept 2011)
15. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G.: Multiobjective exploration of the starcraft map space. 2010 IEEE Conference on Computational Intelligence and Games (CIG) (2010)