Fortress generation for Dwarf Fortress

Procedural Content Generation in Games Autumn 2014

Jakob Melnyk, jmel@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

***Abstract*— Most PCG tools are made with the intention to help the developer reduce costs or be integrated into the games themselves. In this paper we present a PCG tool for generating layouts of fortresses for the game Dwarf Fortress using evolutionary programming. Our results show that our method generates useful maps, but that it is too slow when generating fortresses that can hold a large number of dwarves.**

I. Introduction

This paper was written as part of a project for the "Procedural Content Generation in Games" course on the Games Technology track at the IT-University of Copenhagen. Both the source code[1] and test data[2] for the project is available online.

Games that involve planning the layouts of buildings, rooms, cities, etc. often require the user to plan far ahead in order to create the perfect layout. The complexity of these games can often be so great that new users are put off by the steep learning curve and decide not to play at all. One such game is Dwarf Fortress[1]. Dwarf Fortress focuses on building a large, underground fortress, where dwarves work, eat, and sleep. In order for the dwarves to be effective, the player must create a layout that facilitates effectiveness, e.g. workshops being located close to stockpiles that contain the materials they need. Procedural Content Generation can be used to ease processes such as these and the intent of this paper is to showcase an algorithm that can be used to generate it layout for fortresses in Dwarf Fortress.

The aim of this paper is to generate useful fortress layouts for Dwarf Fortress using evolutionary programming. We begin by giving a short overview of Dwarf Fortress and Evolutionary programming. Then we introduce the method we use to generate layouts and how we evolve the layouts in the search for an optimal layout. After presenting our method, we classify our algorithm using the Procedural Content Generation taxonomy[2]. A report of our results follows the classification and we round out the paper with a discussion of alternative methods and potential future work.

II. Background and Game Design

Procedual content generation (PCG) has been around for a long time, but it continues to evolve and improve as more reason is done in the area. It is used to create content for a game through algorithms. The main benefit of PCG is that it often produces content faster and cheaper than manually creating content.

PCG can also be used to help players. For example, a PCG tool for Minecraft would be able to suggest different ways of building a base to the player, either from the ground or based on something the player had already built. Most PCG tools are made for developers to use in an offline context or as part of the game itself in order to cut down costs. Most games are completely fine without PCG tools for player use, but there are a few games out there where it would benefit the player greatly if there were PCG tools to help in decision making. One such game is Dwarf Fortress.

A. *Dwarf Fortress*

In Dwarf Fortress the player leads an expedition of dwarves in order to create a new home for them. The game takes place in a 3D map (displayed to the player as a 2D map with multiple layers), where the player can direct dwarves to perform various tasks (mining, gathering plants, building furniture, crafting weapons, and so forth). The goal of the game is to build a huge fortress for the dwarves and to keep them alive for as long as possible.

Dwarf Fortress is often described as having a very steep learning curve, as it tells the player nothing about how to play or what they are supposed to do. All of it is something the player has to figure out by themselves. This often means that a player's first fortresses will be of very low quality, as they discover more and more things they have to add to it that they did not plan for.

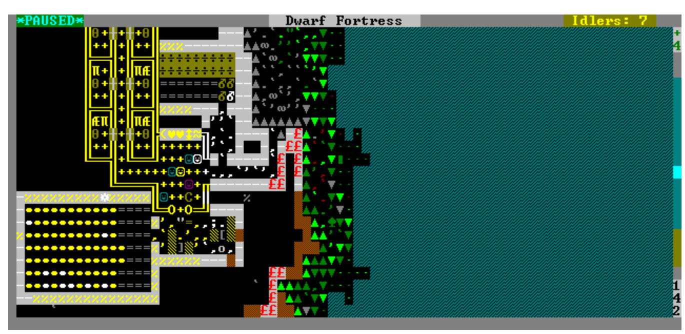

Fig. 1. A screenshot of a very basic fortress.

1) Reducing the problem: Dwarf Fortress contains an immense amount of possibilities. There are 31 workshops[3] from which dwarves can craft different objects, 12 different types of rooms[4] each with their own function and 16

[1]https://github.com/jcgr/MPCG-DFFortressGenerator

[2]https://drive.google.com/file/d/0B-DyxsuStO29cUNteHRKN055a2c /view?usp=sharing

different stockpiles[5] which are used for storing specific types of items. These are all related in different ways and even with just the elements mentioned above, it is already overwhelming for most new players. That is without mentioning special types of constructions, mechanics and the militaristic area of the game.

In order to preserve clarity in the tool, we reduced the amount of possibilities we include. Instead of including all types of rooms, workshops and stockpiles (from now on referred to simply as ”rooms”), we cut it down to the ones we felt were the most essential ones to any fortress. This reduced the total number of rooms from 59 to 23, a much more manageable number when making the interface.

Due to the nature of the algorithm, it is entirely possible to expand the number of room types without significant change in run time. It would simply make the interface of the pool more cluttered.

B. Evolutionary programming

Evolutionary programming[6, Chapter 2] is based on biological evolution. It works on the principle that if the program knows how to evaluate the quality (fitness) of an object and it knows how to create a variation of this object, over time it will be able to search the space of possible solutions for a problem and chose an optimal solution (either a local or global optimum).

Evolutionary programming is often used for optimization problems, as they are quite solid in what results they produce. Assuming that the evaluation function is representative of how an object should be evaluated and the variation operators create the necessary diversity (and thus facilitate novelty), the fitness of a population should continuously improve as the algorithm attempts to imitate real world evolution.

This also applies to fortresses in Dwarf Fortress. As creating a good fortress layout is, in essence, an optimization problem, evolutionary programming is a very suitable technique to use. Compared to doing it ”by hand”, evolutionary programming is able to explore far more possibilities in a much shorter time frame and still arrive at a decent, if not great, result in the end.

III. Methods

There are many different algorithms for dungeon generation[7, Chapter 3], but none that fit our needs specifically. Because we want to generate map layouts where all rooms are connected to the entrance, but not necessarily each other, agent-based dungeon growing did not feel like the right choice. Instead we made the tool using our own map generation algorithm.

The map generation algorithm consists of two different parts: A layout generation algorithm to generate the basic layout of the map (see section III-B) and evolutionary algorithms[6, Chapter 2] to determine how the map layout can be used in the best way (see section III-C).

A. Interface

The interface was created for two purposes: To let the user select options for the map generation easily and to let the user browse the generated maps.

We accomplished this by splitting the interface in two parts. The right part (see figure 3) contains all the options for the map generation. The user can select the dimensions of the map surrounding the fortress to be generated, the number of dwarves that should inhabit the fortress and which rooms that are required to have in the fortress.

The left part (see figure 2) shows an empty box until maps have been generated, at which point it shows the map the user has selected (by default it selects the map with highest fitness). There are two drop-down menus which allows the user to select which of the generated maps they want to see (ordered by descending fitness values) and what layer of the map they are shown.

Fig. 2.

B. Map generation

Our layout generation algorithm is as follows:

1) Create a map of the user-specified dimensions and calculate the number of rooms we want, based on how many dwarves the user expects to have in their fortress.
2) Create an entrance to the fortress on the top layer along one side of the room.
3) For each layer in the map (starting at the top one), make a list that contains all positions that are not dug out and then do the following:
    a) Choose a random position from the list of open positions and use it as one corner of the room.
    b) Pick a random, diagonal direction (north/east, south/east, south/west or north/west), go 6 tiles that way and see if the positions in that direction are open.
    c) If they are open, build an empty room in the middle 4x4 square, set the tiles around that square

Fig. 3.

to room walls, and remove the positions now occupied by the room from the list of open positions.

d) Create a path from the room to the nearest entrance using Breadth-first search[8][9](stairs on the lower levels count as entrances to that level).

e) Once enough rooms have been created, or only 10% of the layer is open, create stairs to the next layer, and connect these stairs to the entrance of the current layer.

f) Repeat steps a) to e) for each layer until enough rooms have been built.

If the algorithm fails to generate a staircase (this is most common when selecting small maps with many dwarves), it will stop generating more rooms and begin doing evolution.

C. Evolution

While evolutionary algorithms work on the same principles, there are some points we feel are important to discuss in our implementation.

1) Initialization of maps: In order to evolve maps we need maps. Prior to the evolution, we generate 10 maps using the map layout algorithm described in section III-B. For every map we generate, we also find and save the distance from every room to every other room, as we need the distance for our objective function. Saving the distances ahead of time allows us to save time when we need the distances as we can simply look them up.

We find the distances between rooms by using breadth-first search[8][9]. Every room finds and saves the distance it has to every other room. In order to save time during this step, we save the direction in both directions when a distance is found (the start room knows the distance to the target room and the target room knows the distance to the start room). Other ways of finding the distances are discussed in V-A.

2) Candidates for evolution: The genotypes (the representations used for evolution) are not the maps that are generated, but rather what is in each room in the map. The problem of evolving optimal map layouts is two-fold: one search space is the placement of each room, e.g. room 01 is located at location (22,13), and the other search space is the combination of room types (”room assignments”) in the layout, e.g. room 01 is a barracks. Our algorithm only mutates room assignments and not room placement, as optimizing in both search space is very expensive.

This means that each map has its own set of room assignments that are not mixed between maps. The evolution runs for 100 generations. Every generation of the evolution the current best room assignment (parent) of each map spawns 100 room assignments (children) that are mutations of the parent. Every room in a child has a 30% chance to be changed into a random room. The child with the best fitness value of these 100 children is then compared to its parent. If the fitness value of the child is better than or equal to the fitness value of the parent, the child replaces the parent as the current best room assignment for that particular map.

Using this method of evolution, we essentially select a random subset of the ”room placement” search space and search the ”room assignment” search space of each element in the subset of ”room placement”. By never changing the original random select of room placements, we ensure that we do not converge towards a local optima too quickly.

3) Objective function: Our objective function works in two steps. First we check if the room assignment being evaluated has all the different types of rooms that the user requested. For each missing room, the fitness is penalized depending on how many generations have already passed (later generations are penalized more heavily).

After that, it calculates the the fitness value of each individual room. This calculation is based on the distance between the room and other rooms it has a relation to. Depending on whether it should be far from, or close to, the other rooms, its fitness is changed.

The relation between rooms is something we have defined based on how we felt rooms related to each other in the game. As an example, it makes sense for a barracks to be near the entrance, as it allows the military to respond to any threat quickly, while the bedrooms should be far from the entrance for exact opposite reason; having monsters barge in on sleeping dwarves is never a good thing.

When relations to other rooms have finished, the function also checks how many we have of that type of room already. If we have more than we expect we will need (varies depending on room type), the room’s fitness is lowered to a factor of $1/2^{numberOfExcessRooms}$. After this is done, the value of the room is added to the room assignment’s total fitness value and the the process starts over with the next room.

4) The algorithm as pseudo-code:

1) Create 10 different maps layout (using the map generation algorithm, see below) with empty rooms. When a map has been generated, find the distance from any room to all other rooms (see III-C.1).
2) For each map layout, create create a random initial room assignment.
3) For every map layout, do the following every generation until enough generations have passed:
    a) Use the current best room assignment as candidate for the generation.
    b) Create 100 mutations of the candidate.
    c) Calculate the fitness value for each room assignment.
    d) If a room assignment is better than its parent, replace the parent with the room assignment.
4) At the end, for every map, copy its best room assignment into the actual map.

D. Taxonomy

Following the taxonomy in [2], we have classified our algorithm as follows:

- **Offline** - Our algorithm is run as part of a tool that does not integrate into Dwarf Fortress[3].
- **Optional Content** - It is up to the user whether they want to use the tool (and thereby the algorithm) or not.
- **Parameter Vectors** - The algorithm takes some parameters from the user (fortress dimensions, number of dwarves and which rooms the user wants).
- **Stochastic Generation** - Most of the generation uses randomness in order to determine what to do.
- **Generate-and-test** - We use evolutionary programming as part of the algorithm, which, by definition, tests everything it generates[4].

IV. RESULTS

In our tool, there are five parameters that can be adjusted (internally, with no interface to the user) in order to influence the evolution:

- **Mutation chance** - The chance that a room is transformed into a random type of room during evolution.
- **Number of children** - The number of children that are spawned for each map every generation.
- **Number of generations** - The amount of generations to evolve the maps over.
- **Missing room penalty** - How much a room assignment is penalized for every required room it does not contain.
- **Missing room penalty scaling factor** - How much the missing room penalty is scaled up for every generation that has passed. The idea is that the closer we get to the end of the evolution, the more important it is to make sure a room assignment had the required rooms.

[3]It is also quite slow which would make it poor for an online tool should it later be interfaced with the game[10].

[4]It does, however, stop after a certain number of iterations and returns the best results.

There were two things we wanted to test in regards to the maps our tool generates:

1) Are the better rated maps actually better than the others? (I.e. does our fitness function work as we would expect it it?)
2) What influence, if any, does changing the parameters have on the quality of the generated maps?

In order to answer both questions, we set up the tool so it could run evolutions with different values for the parameters. We created a combination of each of the following values for the parameters, where the number of generations was equal to $\frac{10.000}{numberOfChildren}$:

Mutation chance	0.1	0.2	0.3
Number of children	10	100	1000
Missing room penalty	10	100	200
Missing room scaling factor	10	100	200

With the number of generations being either 1000, 100 or 10, we had a total of 81 different parameter value combination.

In order to test all combinations equally, we generated 50 maps at the beginning. For each combination, we cleared all 50 maps of their content (meaning that all rooms were empty), loaded the values for the parameters from the combination and then ran the evolution on every one of the maps. At the end of each combination, we saved the generated maps - along with their fitness values - to a unique text file, which we could then examine.

The tests were run on a map with the dimensions 40x40x4 with room for 30 dwarves. We felt that 30 dwarves was a good number, as it is a manageable amount for most players once they have grasped the basics of the game and was within a reasonable time frame for us to test.

The rooms chosen as required rooms were bedrooms, dining rooms, barracks, farms and officies. From workshops we chose the carpenter's, the craftdwarf's, the kitchen and the mason's workshop. From the stockpiles we chose finished goods, food, furniture, stone and wood. All of these are what we feel make up the very essential of any basic fortress.

A. Quality of Maps

Question 1 required us to look through some of the samples and manually compare the best rooms to the other rooms. We compared the best map to the worst map and to the map that was evaluated to be average. Comparing to more maps could provide more precise results, but we felt the trade-off in time was not worth it. This is particular relevant as our manual evaluation metric is subjective, even if this subjectivity is influenced by experience from playing the game.

We looked at 10 of the 81 datasets, all randomly chosen. For all 10, the map that was rated as the best was better than the two we compared it to. Had it not been the case for one or more, we would have looked at more datasets. As we looked at more than 10% of the different datasets and the data confirmed that better maps had higher fitness than worse maps, we believe our fitness function to be effective. Due to the subjective nature of our evaluation (and our objective function), it is possible that the evaluation of maps is not extremely precise. This potential lack of precision is not too big a worry, however, as most users of this type of tool are not likely to use it with the intent for perfection; instead it is a guideline to a layout that can be expanded through gameplay.

In order to answer *Question 2*, we compared the different parameters by keeping them static and comparing the best maps generated where that parameter was the same. For map layouts we used the map layouts for all tests, such that the randomly selected subset of room placements did not influence the change in parameters.

What we learned was surprising. The mutation chance did not have any notable influence on the generated maps. We believe this is due to the fact that we can spawn so many children during evolution (10.000 in total for each room placement), that even with a low mutation chance there is enough time to "catch up" to the variety in children that the higher mutation chances have.

The number of children and number of generations also did not have any notable influence, which is due to the fact that we chose numbers where $numberOfChildren \times numberOfGenerations = 10.000$. Had we chosen test set-ups where the multiplicative of these two parameters did not equal 10.000, it is likely that we would have seen a difference in the quality of the maps.

Only the missing room penalty and its scaling factor had any real influence on the quality of the generated maps. With the values $missingRoomPenalty = 100$ and $missingRoomPenaltyScalingFactor = 10$, most of the required rooms were present in any map. At 200 and 100, respectively, all rooms were present in all of the best maps.

B. Speed

During testing, it became clear that we experienced significant slowdown in generation of map layouts when we added more dwarves.Testing revealed that approximately 90% of the total run time of our algorithm was spent calculating the distances between rooms.

During testing, we noticed that the program ran slower the more dwarves we wanted to generate maps for. To figure out if it was a specific part of the program that was the problem, or if the entire program simply was slow, we timed the different parts of the evolution algorithm.

The reason for this is that we find the distances in a very non-efficient manner. Every room finds the distance to every other room, which has a cost of N^2 where N is the number of rooms. While we cut that down to $N(N+1)/2$ by saving the distance both ways when a target was found, it is still $O(N^2)$.

The problem is that we use Breadth-First Search[8][9] every time we find the distance between rooms. With the way our map works, BFS has a worst case time complexity of $O(4V+V)$ where V is the number of tiles on the map. $4V$ comes from the fact that every tile will check its 4 neighbour tiles to see if they have been visited or not and worst case every tile will have to do so. This leads to a total cost of $O(N^2(4V+V))$, which is an immense cost investment compared to our other functions.

The subject of how to speed up the algorithm is discussed in section V-A.

V. Discussion

There are three main points of discussion on the choices we made with regards to our method: choice of algorithm and method for calculating distances, our objective function, and our mutation method. It is also of interest how the method could potentially be used in generating layouts for fortresses and similar buildings in other game genres.

A. Distance Calculations

As discussed in section IV, the dominating factor on time spent when evolving maps is the calculation of distances between rooms. Because the objective function requires that the distance from every room to every other room is known, these distances must be calculated at some point during the generation of the maps. Our use of breadth-first search in the method described in section III is quite ineffective and has potential for vast improvements.

1) Choice of Algorithm: Because we currently calculate distances individually, the use of a heuristic in our search would highly beneficial. As straight line distance (SLD) would be clearly admissible (no path can be shorter than SLD), we could benefit from the use of A*[11], [12] using SLD as the heuristic. Since A*'s performance is O(E)[13][5] where E is the number of edges, this would reduce the time complexity of the distance calculations to O($N^2(4V)$).

2) Calculation of Distances: While A* would certainly be more effective with our current approach, a different optimization would be to continue using breadth-first search. If we searched for the distance to all other rooms at once instead of individually, we could reduce the current O($N^2(4V+V)$) time complexity to O($N(4V+V)$), which is clearly desirable.

3) Separate Layers: A further improvement would be to only calculate the distance to the entrance and exits of a layer. Whenever distance calculation requires finding the distance to a room on a different layer, it is simple to find the distance to the entrance/exit and add the distance from that tile to the target tile. This would mean that N will be equal to the number of rooms on the layer instead of the total number of rooms in the entire map (significantly speeding up the process when a large number of rooms is required).

B. Alternative Objective Function

An alternative to our current objective function would be to reward novelty. With the current objective function, we encourage sameness as we are unlikely to have a high fitness value if an area contains many areas with no correlation. Adding a bonus to the fitness score of a layout dependent

[5]Assuming optimal heuristic.

on the number of unique rooms it contains would encourage novelty in area assignments.

C. Alternative Mutation Method

Our current mutation method is not very vulnerable to local optima until later on in the generation due to the missing room penalty increasing with each generation. In order to speed up map generation, however, it might be desirable to hit a local optimum earlier. This could be done by mutating area assignments as we do currently, but in addition creating a copy of the assignment with a random ordering, then comparing the two and using the one with the better fitness value when comparing to the parent. This would test two area assignments per mutation, which would then allow for a reduction in the number of generations done in evolution. However, fewer generations would means fewer alternatives being tested, which could potentially result in the evolution getting stuck in a local optimum.

D. Generalization To Other Game Genres

While the method is not likely to be useful in an online context, as it is simple too slow (online PCG must be fast[10]), it could be useful for suggesting a layout of a castle, fortress or other similar structure to a level designer looking for inspiration. Such a change would require a different interface that could take into account the needs for specific types of rooms (and the weights of their relation), but the method itself would still be fully functional.

E. Future work

Besides the improvements mentioned in the discussion above, there are a number of smaller things we would consider improving should we do any future development on the tool.

We currently assume that all rooms will be four by four tiles in size. This is rarely the case in actual play, so changing the size of a generated room depending on the room type would be improve how well the output of the tool represents the actual game. It does, however, introduce the problem of how to generate the map layouts with a variable room size. This could potentially be solved using a lower and an upper-bound on room sizes.

The method for placing rooms described in section III concentrates the placed rooms on the upper layers of the map. This is not always desirable, as the lower levels of the map will be unused. It also leaves little room for expansion on the upper layers. A potential solution to this problem would be to place an equal percentage of rooms on each layer selected by the user. This would ensure a somewhat even distribution of rooms on all layers and would reduce concentration of rooms. This could lead to longer paths in general, but this should not result in worse performance if the distance calculations is separated into layers as suggested in section V-A.

Because the tool is currently not able to show more than 50 tiles on the vertical axis, some of the map is cut out when generating large maps. This is impractical for the user, so adding a scroll function (or similar) to the map window would be a definite improvement to the tool.

In addition, the map is currently displayed in ASCII characters. This is, at best, not very readable as the characters are taller than they are wide. Changing the ASCII characters into the characters used by the real game or to another graphical representation would improve the usability of the tool.

Adding a save function to the layout generation tool could be beneficial to a user, as it is currently necessary to record the maps, if they must be saved for later use, e.g. by taking a screenshot of each layer.

F. Contribution to the project

Most of the work done on the project was shared equally between both of the authors, but there were some areas which were primarily done by one author.

Generation of initial map layouts and the user interface was primarily programmed by Jacob Grooss. Evolution of the maps and the distance calculations were primarily programmed by Jakob Melnyk.

Sections II, III, and IV of the paper was primarily authored by Jacob Grooss. Sections I and V as well as the abstract was primarily authored by Jakob Melnyk.

REFERENCES

[1] T. Adams and Z. Adams, "Dwarf Fortress," 2006, accessed December 12, 2014. [Online]. Available: http://www.bay12games.com/dwarves/

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation," in *Applications of Evolutionary Computation*. Springer, 2010, pp. 141–150.

[3] D. F. Wiki, "DF2014: Workshop," accessed December 12, 2014. [Online]. Available: http://dwarffortresswiki.org/index.php/DF2014:Workshop

[4] ——, "DF2014: Rooms," accessed December 12, 2014. [Online]. Available: http://dwarffortresswiki.org/index.php /Category:DF2014:Rooms

[5] ——, "DF2014: Stockpile," accessed December 12, 2014. [Online]. Available: http://dwarffortresswiki.org/index.php/Stockpile

[6] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Springer, 2007.

[7] J. Togelius, N. Shaker, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2014.

[8] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[9] U. of California, "ICS 161: Design and analysis of algorithms, lecture notes," 1996, accessed December 12, 2014. [Online]. Available: http://www.ics.uci.edu/ eppstein/161/960215.html

[10] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based Procedural Content Generation: A Taxonomy and Survey," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 172–186, 2011.

[11] P. Lester, "A* Pathfinding for Beginners," 2005, accessed December 12, 2014. [Online]. Available: http://www.policyalmanac.org/games /aStarTutorial.htm

[12] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths."

[13] S. Russell and P. Norvig, *A*.