Fortress generation for Dwarf Fortress

Procedural Content Generation in Games Autumn 2014

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

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.

For the project we could do one of two things: Develop a new PCG algorithm, or apply an existing algorithm (or combination of algorithms) to some game domain in a novel way.

A. Problem Statement

For our project, we decided to generate fortresses for the game Dwarf Fortress[1] by creating random map layouts and and then evolve the contents of the maps.

Our goal was to be able to generate fortresses that could be used in-game in practice, even though they may not be perfect fortresses[2]. We decided to not aim for perfect fortresses, as Dwarf Fortress is an amazing complex game and there are an immense number of variables to take into account in order to create a perfect fortress.

II. Background and Game Design

Procedual content generation has been around for a long time, but it continues to evolve and become better. As it is now, it is mostly used by game creators to create content for their games. There are, however, some games where a PCG tool will be able to help the players learn more about the game and how to do better while playing.

Stuff about pcg/evolution

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 on 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 them self. 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.

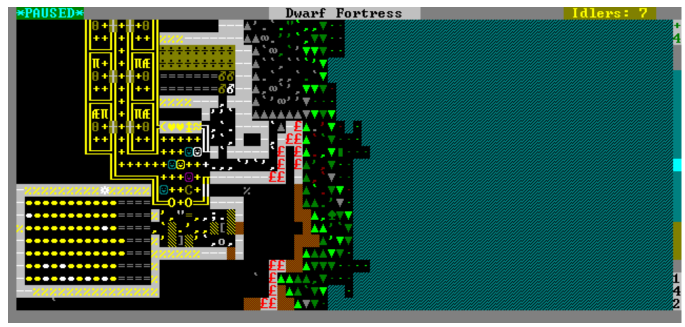

Fig. 1. A screenshot of a very basic fortress, which is confusing for anyone who does not know how DF works.

In order to help the novice player out, we created a fortress generator that can give them some basic layouts from which they can choose which one they like the most. These layouts also give the player an idea of how many different things they need to play for in any future fortresses they may want to make.

1) Reducing the problem: Dwarf Fortress contains an immense amount of possibilities. There are 31 workshops from which dwarves can craft different objects, 12 different types of rooms each with their own function and 16 different stockpiles which are used for storing specific types of items. These are all connected in different ways and even with just the mentioned things, it is already overwhelming for any new player. 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, we cut it down to the ones we felt were the most essential ones to any fortress. This reduced the total number from 59 to 23 different rooms, workshops and stockpiles, a much more manageable number.

It should be noted that adding some of the things that we left out, would only have a minimal influence on the time it takes to generate our maps.

[1]http://www.bay12games.com/dwarves/

[2]"Perfect" in this context meaning a fortress that encapsulates everything a fortress can have, with a layout that optimizes everything.

B. Evolutionary programming

Evolutionary programming[1, Chapter 2] is based on biological evolution. It works on the principle that if the program knows how to evaluate if an object is good and it knows how to change this object in order to affect how good/bad it is, then it should be able to create a good result given enough time.

Evolutionary programming (EP) is often used for optimization problems, as they are quite solid in what results they produce. Assuming that the evaluation function is well written, the evolution should keep moving towards better results, without the need for humans to constantly change things.

This also applies to fortresses in Dwarf Fortress. As creating a good fortress layout is, in essence, an optimization problem, evolutionary computing is a very suitable technique to use, as it is able to explore a lot of possible fortress layouts that would not be feasible by ”hand”.

Need some sort of ending

III. Methods

There are various different ways to generate dungeons, some are discussed in [2, Chapter 3]. Depending on the type of dungeon one wants **needs more stuff**

For our dungeon generation, we use two different algorithms: A map generation algorithm to generate the basic layout of the map (III-C) and evolutionary algorithms[1, Chapter 2] to determine how the map layout can be used in the best way (III-D).

A. Taxonomy

B. Interface

The interface for our tool is simple, as we want to avoid confusing the user. On the right is has the options for what the map should include and the dimensions of the map. On the left, the generated maps are shown.

insert UI picture(s) here

C. Map generation

For the map generation algorithm, we decided to write our own algorithm, as we could not find any other algorithm that could do specifically what we wanted. Our map generation algorithm works 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:

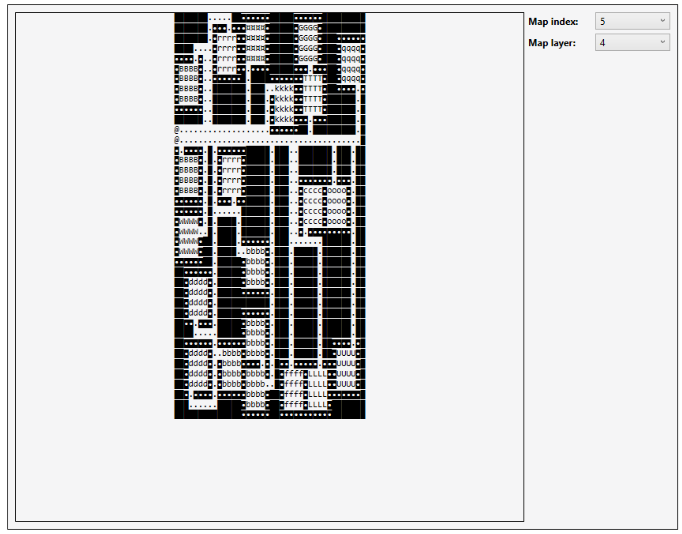

Fig. 2.

   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 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 Dijkstra (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.

D. Evolution

While evolutionary algorithms often work in the same way, 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. At the beginning of the run, we generate 10 maps using the map generation algorithm described in section III-C. 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 Dijkstra’s algorithm[**?**][**?**]. Every room finds and saves the distance it

Fig. 3.

has to every other room. In order to save time during this step, we save the direction in 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: We decided that our genotype (the thing that is evolved) should not the maps that are generated, but rather what is in each room in the map. We do not want to risk throwing a map out due its fitness being low, as it is possible that another genotype (which we will refer to as 'room assignment') for a map could be good.

This means that each map has its own set of room assignments that are not mixed between maps. For each generation in the evolution, a number of children are spawned from the currently best room assignment. For every new child, every room had a 30% chance to be changed into a random room. After the mutation is finished, the best child will become the parent for the next generation assuming it is better than the parent it was spawned from.

This way of doing the evolution let us keep all the map layouts we generated at the beginning, while still being able to evolve them in order to reach better ways of building the fortress.

3) Objective function: When the user starts the generation, they can select which rooms they want in their fortress (see section III-B). If a room assignment that does not contain the chosen type of room, or enough of certain rooms (there are some the user will need more than one of, for example bedrooms and dining rooms), the room assignment will be penalized based on how many that are missing.

After the penalizing of missing rooms, the objective function iterates over every room and checks its distance to rooms that it has a relation to (a barracks should be close to the entrance, for example). Depending on the distance, the room is either penalized or rewarded. This is where calculating the distanced ahead of time (see III-D.1) saves a lot of time, as we have all the values already. All we need to do is figure out which rooms are related.

Penalize more than the required number of rooms

Need something about the relationships between rooms

4) Maps, room assignments and generations: We decided to generate 10 different maps for the evolution. We kept the number fairly low, as **needs ending**.

For each generation, we decided to create 100 room assignments for every map. This gave us a huge variety of room assignments, which made it likely that at least one was better than its parent.

On top of that, we decided to evolve our maps over 1000 generations, as that guaranteed that we got to explore a lot of different layouts.

5) The algorithm as pseudocode:

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-D.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.

IV. Results

In our tool, there are five parameters that can be adjusted in order to influence the evolution: The chance that a room is mutated; the number of room assignments that should be generated for a map during evolution; how many generations the evolution contains; how much a room assignment is penalized for not having the chosen rooms; and how much higher this penalty is the further the evolution progresses.

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 $\frac{10.000}{numberOfRoomAssignments}$:

Mutation chance	0.1	0.2	0.3
# of room assignments	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 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.

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 room to the worst room and to the room that was evaluated to be in the middle of the maps. Comparing to more maps would provide more precise results, but we felt the trade-off in time was not worth it.

We looked at 10 of the 81 datasets, all randomly chosen. For all 10, the room 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. We believe, as we did not find any proof saying otherwise, that the batter rated maps are actually better than the other maps in most cases. It is possible that some maps are not rated precisely enough, but it is few enough cases that it should not be an inconvenience to the user.

In order to answer *Question 2*, ran an iterative process where we selected a certain parameters they should have in common (for example evolution chance) and compared them to other datasets where the other parameters varied. Whenever the maps in most of a dataset had changed in a notable way (more/less of the required rooms, more scattered rooms and so on), we would figure out why the had happened.

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 create so many room assignments in total during evolution that even the low mutation chance has a chance to "catch up" to the higher one's assignment variety.

The number of room assignments and number of generations did not have any notable influence either, which is due to the fact that we chose numbers for each where $numberOfRoomAssignments \times numberOfGenerations = 10.000$. Had both numbers been significantly lower, the quality of the maps may have been decreased.

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

B. Speed

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.

While we do not have any specific timings (due to the fact that different machines run faster/slower than each other), it was clear that the distance finding between rooms was the slowest. It took approximately 90% of the total run time, even when we ran 1000 generation with 100 room assignments per map.

The reason for this is that we find the distances in a very non-efficient manner. Every room is told to find 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 every time we find the distance between rooms. With the way our map works, BFS has a worst case cost 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 $O(N^2 \times (4V + V))$, which is an immense cost investment compared to our other functions.

The subject of how to speed up the distance finding 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*[3], [4] using SLD as the heuristic. Since A*'s performance is O(E)[5][3] 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 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[6]), 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

[3] Assuming optimal heuristic.

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.

REMEMBER TO REMOVE NOCITE FROM REFERENCES

REFERENCES

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

[2] 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.

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

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

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Pearson Education Limited, 2011.

[6] 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.

[7] R. Poli, W. B. Langdon, N. F. McPhee, and J. R. Koza, *A Field Guide to Genetic Programming*. Lulu. com, 2008.

[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 11, 2014. [Online]. Available: http://www.ics.uci.edu/ eppstein/161/960215.html

[10] T. Adams and Z. Adams, "Dwarf Fortress," 2006, accessed December 11, 2014. [Online]. Available: http://www.bay12games.com/dwarves/

[11] D. F. Wiki, "DF2014: Workshop," accessed December 11, 2014. [Online]. Available: http://dwarffortresswiki.org/index.php/DF2014:Workshop

[12] 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.