# 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

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

**http://en.wikipedia.org/wiki/Dwarf_Fortress**

Dwarf Fortress is a game about leading an expedition of dwarves in order to create a new home for them. The game takes place on a procedurally generated 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.

[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.

Dwarf Fortress is often described as having a very steep learning curve, as it tells the player nothing about how they play or what they are supposed to do. All of it is something the player has to figure out by themself. 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.

In order to help the novice player out, we intend to create a fortress generator that can give them some basic layouts from which they can choose which one they like the most. These layouts should 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.


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

### B. Evolutionary computing

Evolutionary computing[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 computing (EC) 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. Game Design

Whats the design of the game you are going to use? Why do you need PCG in this game?

Design: Tile-based map where you are the leader of a dwarven expedition. You tell the dwarves what to do and attempt to build a huge fortress in which they can live and thrive.

PCG: Because coming up with a fortress layout can be very difficult, especially for new players. Having an idea of how to create a fortress would be an immense help.

**Ikke sikker p dette kapitel er ndvendigt. Er ikke i eksempel rapporten vi har fet. Muligvis copy/pasta fra Modern AI, men str inde p PCG siden p Learnit**

## IV. Methods

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

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

### A. 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**

### B. 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:
    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.

### C. 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 first need the maps. At the beginning of the run, we generate 10 maps using the map generation algorithm described in section IV-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.

During our testing, calculating distances between rooms was the slowest part of the entire process by far. This is due to two things: We use Dijkstra's algorithm[3] instead of the A* algorithm[4], which in many cases is slower. The other thing is that for each room we want to find distances from, we look for another specific room. When that room is found, we save the distances (for both rooms, as the distance is the same the other way). When looking for such a specific target, A* would have been a better choice and is discussed in detail in section VI-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 IV-A). 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 IV-C.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.

**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 IV-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.

## V. Results

Did it work? How well? Provide some figures, and a table or two. How much time does it take? Remember to include significance values (remember the t-test?), variance bars Reread some of the papers from class and compare how they report their results.

PCG Book 2.4: Evaluation functions

PCG Book 12: Evaluating Generators

## VI. Discussion

### A. Finding distances

*1) Dijkstra vs. A*:* As mentioned in IV-C we use Dijkstra's algorithm for our distance- and path-finding. We started out using Dijkstra during map generation for creating the path from a room to the entrance of the layer, as we did not know the position of the entrance in the first iterations. This was changed in later iterations, but we did not notice anything that was slower than we expected and therefore did not feel a need to change Dijkstra into A*.

That being said, A* would have improved the speed of the algorithm to some degree, as A* is never worse than Dijkstra as long as the heuristic for A* is admissible.

The point where A* would have made a huge impact is at the beginning of our evolution where we find the distances between all rooms for a given map. We find the distances by iterating over every room and finding its distance to each other room (assuming the distance is not known already) by using Dijkstra. When a room is found, the distance is updated both ways (from the start room to the target and from the target to the start room), which saves us some time.

We still have to run $rooms^2$**check math** pathfindings where we know exactly where our target is. In such cases, A* is generally faster than Dijkstra or, in the worst case, as slow as Dijkstra. We have not been able to find any tradeoffs from using A* instead of Dijkstra.

*2) Other ways of finding distances:* Apart from using A*, there are two other methods we believe would have increased the speed of calculating distances.

The first method would only work for finding the distance between rooms on different layers. Assuming every room knows the distance to the entrance of their layer (stairs leading up) and the distance to the exit of their layer (stairs leading down), we can find the distance rooms on different layers like this: Start with the distance from the start room to the stairs that lead to the next layer. If we are still not on the correct layer, add the distance from the entrance to the current layer to the exit of the layer. When the correct layer is found, add the distance from the end room to the entrance of the layer.

This method is not useful for small fortresses, as they typically are not generated on multiple layers, but for larger fortresses that do span multiple layers, it has the potential to save a lot of time.

Other way is to find distance to all rooms at the same time.

### B. *Future work*

Variable room sizes

## 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] E. Hazzard, "Dijkstra's algorithm - shortest path," 2014. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/dijkstras-algorithm-shortest-path-r3872

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