

A3.6 Maze Generation

WOUTER SLEGGERS
REPORT

Uppsala University
January 7, 2021

19950825-1957
wjslegers@gmail.com

INTRODUCTION

I have elected to do project A3.6 and make a maze generation algorithm. I came up with three different algorithms. The first a search algorithm, the second is inspired by random walks and the last one is inspired by evolutionary teaching of AI to play games. The latter is not very suitable for the purpose nor recursive but fun to work with!

Practical information. I use the word maze for any set of fields arranged in a square, with walls around the edges and each field is connected to its horizontal or vertical neighbours either by a 'wall' or an 'opening'. We may speak of fields being connected or not depending on a wall being there. Also I use the words node and field interchangeably. I set the goal of a 'correct maze' as a maze that has a path of connections from the top left to the bottom right. In the code that is from field (0, 0) to field (`maze.fields - 1`, `maze.fields - 1`), where `maze.fields` is the number of fields of maze. The goal is to create an algorithm that makes such a 'correct maze', preferably with as many walls still standing, one could after all just remove all walls and be done with it, but be left with a rather boring maze.

To make writing code as easy as possible I chose Python, as I find it quick and easy to set things up. You can find all the Python files in the zip-folder. I made a 'Maze' class and print it using the 'print()' function of Python to simplify the creating of the graphical part of the program.

1. PATH THROUGH MAZE ALGORITHM

The idea of the algorithm is to first generate a relatively dense random maze. For example, for each wall there is a 70% chance it becomes a wall and 30% it is left open. Then we use a search algorithm that saves all the nodes it can reach from the starting node in a list called `searched`. Every node that is a dead end it also saves in a list called `dead_ends`, a dead end we define as a node with three or more walls around it. If we come across the end node we stop the algorithm. Otherwise we go through the dead ends until we find one that has a wall that leads to a previously unexplored node i.e. the neighbouring node is not in `searched`. We remove the wall and the algorithm repeats the search but starting from the node that we just connected. We can keep the `searched` and `dead_ends` lists, the latter minus the newly opened up field, of course. This way we 'search' each node only once and the time the algorithm takes stays limited.

1.1. Why it works. The idea is that the algorithm will not stop searching the maze and opening up connections until there is a path that reaches the end ensuring that it gives a correct maze. However, it is possible that there are no suitable dead ends during one of the steps in the algorithm. See the figure below with such an example. On the right is the same maze but with the fields marked 's' for searched or 'd' for searched and a dead end. The guarantee we do have is that if we do not reach the end of the maze with our search there must be a searched node with a neighbour it is not connected to yet. So if we do not find a suitable dead end we take such a searched node to open up a new connection from and we continue our search from there. This is necessary to make sure we always end up with a correct

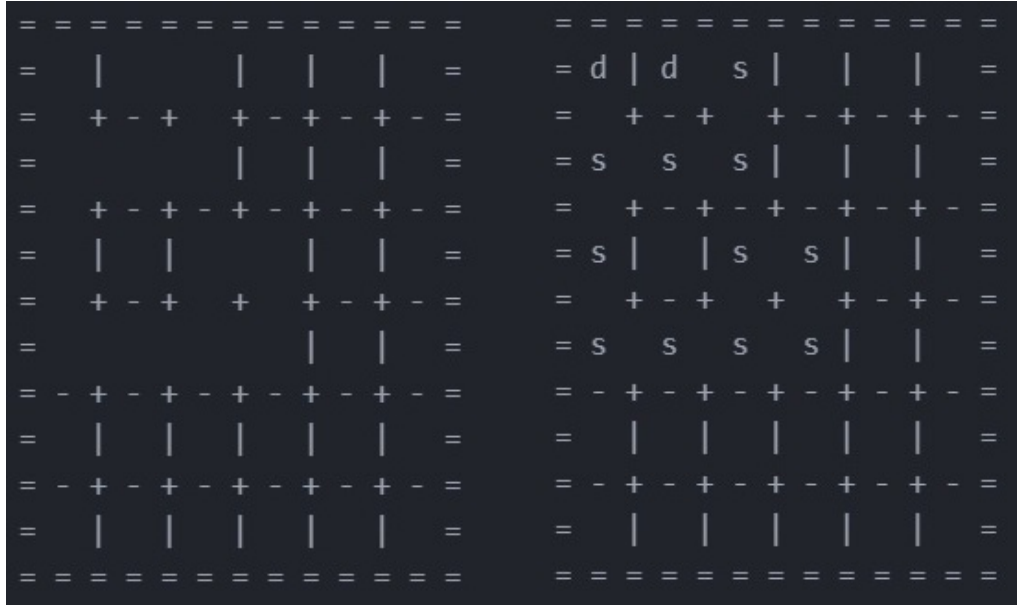


FIGURE 1. Example of a maze without suitable dead ends

maze, but it must be said that the situation does not occur all that often, in part because we start with a randomized maze.

1.2. Tweaks to the algorithm. There are some choices we can make. An important one is the begin density of the maze. Make it too open and there'll already be some paths to the end, and a sparse maze isn't the most interesting. Make it too dense and then there will be whole sections of the maze completely blocked off and never having been searched. The choice of 70% seems to work most of the time. Furthermore we have a choice of which dead end we open up in each step (or alternatively which searched node we open up) and a choice of which neighbour we open it up to. I chose to favour opening up dead ends further away from the diagonal line between the beginning and end, to encourage creating longer paths, with the following line of code.

```
dead_ends.sort(reverse=True, key=lambda node: abs(node[0] - node[1]))
```

After this we take the first suitable dead end we find in the list. For the choice of neighbour we go with a random one. Also if we need to open a searched node instead of searched dead end, it is picked randomly.

1.3. Notes and performance of the algorithm. Since in the worst case each field has to be searched we may have to search n^2 fields for a square maze of n by n . In the worst case each search adds only one node to the list **searched** and each time we do a sort of **dead_ends** (which has time complexity $n \log n$) and in the worst case about a third of the field could be dead ends. These situations are thankfully rather unlikely but, long story short, the worst case time complexity is not great. In practicality it is perfectly capable of creating mazes of 50 by 50 within seconds (on my PC anyway). Going higher than this may lead to call-stack errors, with too many recursive calls taking place.

Unfortunately with the random nature of the algorithm it can happen that few nodes get searched, especially in large mazes. Sometimes less than half gets

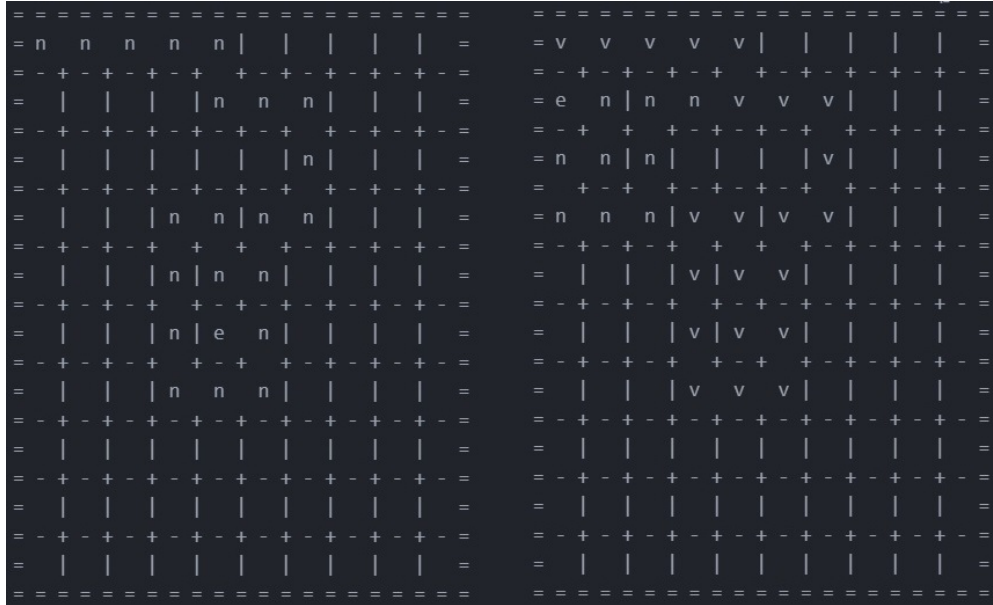


FIGURE 2. Two steps of the walker maze generation algorithm creating a 10 by 10 maze

searched, leaving the rest of the maze as it was randomly initialised and possibly unreachable from the beginning field.

To run the generation just execute `PathThroughMaze.py`. The variables `SIZE_MAZE` and `BEGINNING_DENSITY` can be played around with. Whether we print the steps in between can be changed with the boolean `PRINTING_STEPS`. When printing the maze you can use `maze.print(False)`. False indicates that you want to leave the fields empty. With True it also prints the fields, which during the execution of the algorithm get marked with 's' if they've been searched, which in turn can get overridden by 'd' for dead ends or 'O' if it is a node that has been opened.

2. SELF-AVOIDING RANDOM WALK

The idea of this algorithm is to start a self-avoiding random walk from the beginning of the maze, leaving a path of connections through an initially fully walled in maze. Whenever it can make no further steps we repeat, but start from a previously visited node with a previously unvisited neighbour. We don't stop until every node has been visited. This results in a fully connected maze, which ensures a path from beginning to end. In the figure below we give an example of the first two steps in the algorithm. When the boolean `PRINTING_STEPS` is True we mark newly visited nodes with 'n', the node where the latest path ends is marked with 'e' and nodes visited in previous steps are replaced by 'v'.

Notes on the algorithm. To run the generation just execute `Walker.py`.

The algorithm is the quickest of the three, since nodes get visited once, and not many other calculations happen.

IMPORTANT NOTE: This algorithm can not generate mazes larger than 32 by 32. Since each step it sets means a recursive call we would get a call-stack error. an improvement could be to program the algorithm to only do a recursive call every time it starts a new path, and save visited nodes and the current position and direction in a different way to avoid the problem and create bigger mazes.

3. EVOLUTIONAL LEARNING

I considered suggesting a project on evolutionary learning of neural networks, but instead ended up doing the maze generation project. However, borrowing from that I had an idea for another maze generation algorithm.

We create a certain amount of random mazes. We give each a score and take the best mazes from this generation to go to the next generation. Each of those mazes creates a few mutations of itself to also be tested in this generation. Like in natural evolution, if the random mutations turn out to be beneficial they 'survive' to the next generation.

This requires a lot of tuning of parameters, such as how the mutations happen and most importantly how the mazes get scored.

3.1. Performance of the algorithm. This is a very memory and computationally expensive way of creating mazes, so not the most practical. To create good mazes it seems to help to have some sort of search algorithm integrated in the scoring function. But since the scoring function gets called so very often, that makes this method quite computationally expensive. Furthermore there is no guarantee the algorithm finishes with a maze that has a path from beginning to finish. However by strongly increasing scores for mazes that have a path from start to finish and running enough generations we can make it extremely likely that it does deliver a correct maze.

3.2. Notes on the algorithm. I made two versions, `Evolutionary.py` and `EvolutionaryLongPath.py`. The first uses a scoring function that creates a list of nodes that can be reached from the beginning node, using the `distance_to_end` function. If none of the reachable nodes are close to the end the score will be low. Also a high number of nodes that can not be reached decreases the score. To prevent making the maze too sparse the score is increased slightly for each wall it has. The downside of this method is that the maze it produces often has a rather short path to the end. The other version, `EvolutionaryLongPath.py`, is meant to prevent that. Here, in the scoring function, we use a search function `shortest_path` that keeps track of shortest distance from the start node to each node in the maze. This way we can also punish for nodes that can not be reached and reward for reaching the last node. But now we can also give higher scores to mazes that have a long path to the end instead of a short one! This creates, what I consider to be, more interesting mazes. Again we give a slightly higher score for mazes with more walls. Once a maze is found that reaches the end, the large majority of the mutated children are worse mazes, and the progress becomes very slow. So to fully utilise this and give it more time to create a longer path we can use more generations.

The second version uses a recursive search algorithm that could potentially go through every node and even the same node multiple times if it finds a shorter path there. So for mazes larger than 30 by 30 there is potential for call-stack errors, however this type of algorithm is unsuitable for large mazes anyway since it would take a lot of generations to find a good maze. The latter algorithm in particular is unsuitable for large mazes since the score does not account for being closer to making a path to the end like the first version does. In testing both versions tend to create decent mazes of size 12 by 12 in as few as 30 generations, with 80 mazes in each generation.