AI for Game Programming 2 (5SD810) Project A3.6: Maze Generation

Wouter Slegers January 11, 2021 19950825-1957
Report Uppsala University 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 evolutional teaching of AI to play games. The latter is not very suitable for the purpose nor recursive but fun to work with!

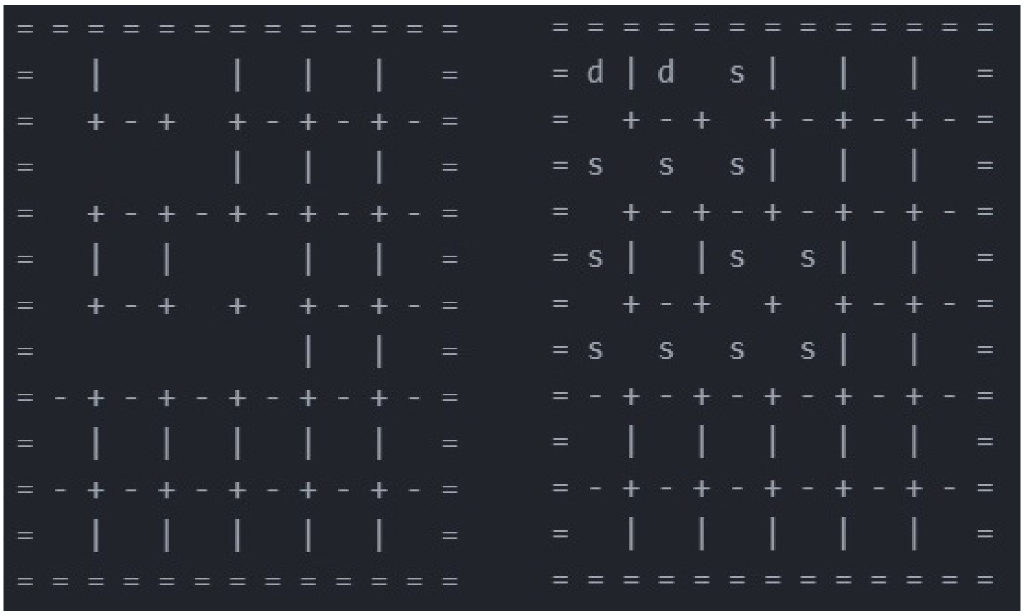
Practical information. I use the word maze for a set of fields arranged in a square, with walls around the edges and each field is connected to its horizontal or vertical neighbours either by a 'wall' or an 'opening'. We may speak of fields being connected or not depending on a wall being there. Also 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. 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 make a connection and the algorithm repeats the search from the new node. We save the lists and use them again in the next step, so that we only 'search' each node once. We end when all the nodes have been added to the list `searched`.

1.1. **Why it works.** The idea is that the algorithm will not stop searching the maze and opening up connections until the maze if fully connected 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 figure 1 for 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. When this situation arises we take a randomly selected previously searched node, that is not necessarily a dead end, but has an unsearched neighbour. Now the algorithm will not stop until it produces a fully connected maze.



```
= = = = = = = = = = = = =    = = = = = = = = = = = = =
=   |       |   |   |   =    = d | d   s |   |   |   =
=   + - +   + - + - + - =    =   + - +   + - + - + - =
=           |   |   |   =    = s   s   s |   |   |   =
=   + - + - + - + - + - =    =   + - + - + - + - + - =
=   |   |       |   |   =    = s |   | s   s |   |   =
=   + - +   +   + - + - =    =   + - +   +   + - + - =
=               |   |   =    = s   s   s   s |   |   =
= - + - + - + - + - + - =    = - + - + - + - + - + - =
=   |   |   |   |   |   =    =   |   |   |   |   |   =
= - + - + - + - + - + - =    = - + - + - + - + - + - =
=   |   |   |   |   |   =    =   |   |   |   |   |   =
= = = = = = = = = = = = =    = = = = = = = = = = = = =
```

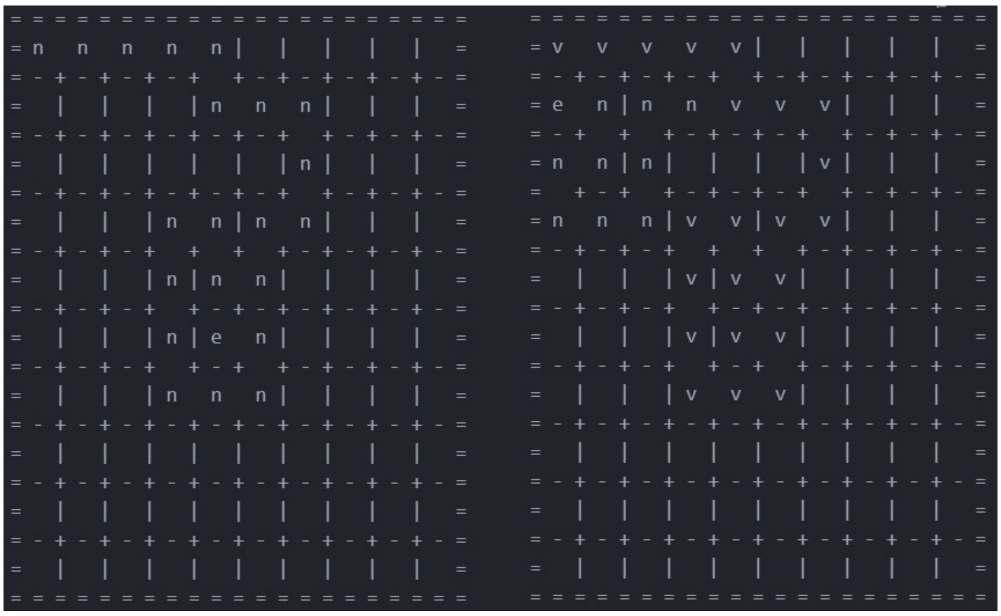
Figure 1. Example of a maze without suitable dead ends

1.2. **Tweaks to the algorithm.** There are some choices we can make. 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.

Another important choice 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. Choosing density 1, i.e. all walls are up, this algorithm becomes more or less a random walk algorithm due to the choices above. However, each step is a recursive call so the size of the maze would have to stay under 31 by 31 to steer clear of Python's default recursive call limit of 997.

1.3. **Versatility of the algorithm.** This algorithm can, instead of starting with a randomized maze, also be applied to any existing maze! If the maze is already fully connected it would remain unchanged, however if it was not then this can ensure it becomes fully connected whilst making as few alterations as possible.

1.4. **Notes on the algorithm.** To run, just execute `PathThroughMaze.py`. The variables `SIZE_MAZE` and `BEGIN_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.



```
= = = = = = = = = = = = = = = = = = = = =    = = = = = = = = = = = = = = = = = = = = =
= n   n   n   n   n |   |   |   |   |   =    = v   v   v   v   v |   |   |   |   |   =
= - + - + - + - +   + - + - + - + - + - =    = - + - + - + - +   + - + - + - + - + - =
=   |   |   |   | n   n   n |   |   |   =    = e   n | n   n   v   v   v |   |   |   =
= - + - + - + - + - + - +   + - + - + - =    = - +   +   + - + - + - +   + - + - + - =
=   |   |   |   |   |   | n |   |   |   =    = n   n | n |   |   |   | v |   |   |   =
= - + - + - + - + - + - +   + - + - + - =    =   + - +   + - + - + - +   + - + - + - =
=   |   |   | n   n | n   n |   |   |   =    = n   n   n | v   v | v   v |   |   |   =
= - + - + - +   +   +   + - + - + - + - =    = - + - + - +   +   +   + - + - + - + - =
=   |   |   | n | n   n |   |   |   |   =    =   |   |   | v | v   v |   |   |   |   =
= - + - + - +   + - + - + - + - + - + - =    = - + - + - +   + - + - + - + - + - + - =
=   |   |   | n | e   n |   |   |   |   =    =   |   |   | v | v   v |   |   |   |   =
= - + - + - +   + - +   + - + - + - + - =    = - + - + - +   + - +   + - + - + - + - =
=   |   |   | n   n   n |   |   |   |   =    =   |   |   | v   v   v |   |   |   |   =
= - + - + - + - + - + - + - + - + - + - =    = - + - + - + - + - + - + - + - + - + - =
=   |   |   |   |   |   |   |   |   |   =    =   |   |   |   |   |   |   |   |   |   =
= - + - + - + - + - + - + - + - + - + - =    = - + - + - + - + - + - + - + - + - + - =
=   |   |   |   |   |   |   |   |   |   =    =   |   |   |   |   |   |   |   |   |   =
= = = = = = = = = = = = = = = = = = = = =    = = = = = = = = = = = = = = = = = = = = =
```

Figure 2. Walker algorithm: Two first steps of the walker algorithm creating a 10 by 10 maze

2. Self-avoiding random walk

When I worked on this second algorithm I must have forgotten that a very similar method 'hunt and kill' was already discussed in the slideshow linked in the project description together with the variation 'recursive backtracker'. I would still like to include my version here, but I will keep it short! 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 from a previously visited node that has an unvisited neighbour. We don't stop until every node has been visited. This results in a fully connected maze. In figure 2 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 algorithm just execute `Walker.py`. Since nodes get visited once, and not many other calculations happen, the algorithm is fairly quick. It always gives a fully connected maze and the paths are, as you might suspect, rather random.

Each path created is another recursive call, so creating very large mazes may have you reach the maximum recursion depth, but it is still faster and capable of much larger mazes than the path-through-maze algorithm with begin density 1.

3. Evolutional learning

I considered suggesting a project on evolutional learning of neural networks, but instead ended up choosing 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 create a few slightly mutated copies of itself to also be tested in that next 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, 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 correct mazes it seems unavoidable 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 that 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. To make it likely that it becomes fully connected however tends to take a lot of generations. Therefore we run our path-through-maze algorithm on the resulting maze just in case it was not already fully connected, or worse, did not have a path to the end. It tends to be likely that one or two fields, throughout the generations, continue to be fully surrounded by walls. This would likely sort itself out after running more generations but running the path-through-maze algorithm does not take long, especially on a nearly finished maze. And thankfully running the algorithm at the end will not make the maze worse, as it only opens up connections to previously unreachable nodes.

3.2. **Notes on the algorithm.** My first version, `Evolutionary.py`, is capable of making fully connected mazes with a path from beginning to end, but often the path from beginning to end was rather short. Hence the final version became `EvolutionaryLongPath.py`.

The scoring function uses a search function `shortest_path_to_field` that calculates, for every node of the maze, the shortest path through the maze from the starting node. This we can use to decrease scores for mazes that have nodes that can not be reached and increase the score if there is a path to the last node. However we can also give higher scores to mazes which have a longer path to the end instead of a short one! This creates, what I consider to be, more interesting mazes. We slightly increase the score for each wall the maze has so it does not become sparser than it has to. Lastly, as long as there is not a path to the end we calculate how close the maze is to creating a path. This we do by calculating the shortest Manhattan-distance between the nodes that can be reached from the beginning and the end. The longer this shortest distance to the end is, the lower the score. This helps finding a maze with a path to the end quicker, especially useful in larger mazes.

The search algorithm that the scoring functions uses potentially has to go through every node in the maze 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 recursion call depth 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 and each generation would take longer too, computationally.

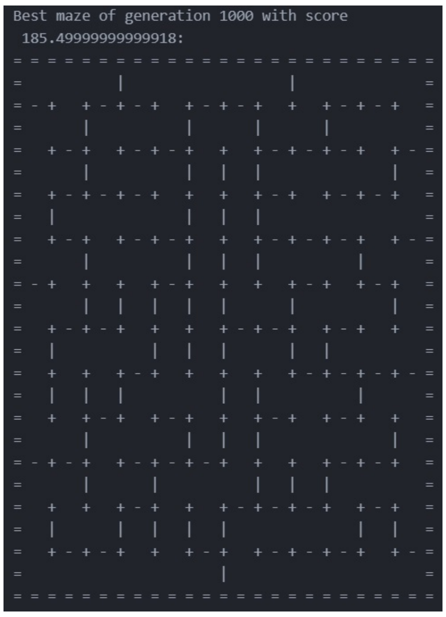
3.3. **Testing results.** In testing the algorithm tended to create decent mazes of 12 by 12 in as few as 50 generations, with 120 mazes in each generation. In testing this I never got a maze that had no path from beginning to end but they were often not fully connected yet. Once a maze is found that has a path from beginning to end end, the large majority of the mutated children are worse mazes, and the progress becomes slow.

Despite it being more taxing on memory, I decided to use multi-processing so I could more quickly experiment with higher numbers of generations. In figure 3 for example is a maze of 12 by 12 after 1000 generations (so you don't have to, each such test tends to take at least a full minute). The best maze of each generation did not change after generation 600 because improvements became too unlikely. You can however see that small improvements are still possible, but the chance of a mutated maze making that improvement and not also seriously lowering the score by removing another wall is extremely small.

Therefore I added a different type of mutated child. These only differ from the parent by mutations in a randomly selected region. It indeed seems to increase the chance of making improvements in later generations and leads to higher scores after lot of generations, but only using these type of children tends to lead to slower progress in the beginning. I settled on using some of each on each generation and left this as another parameter.

Another parameter is the initial density of the mazes we start with. After enough generations mazes tend to converge to similar mazes regardless of this starting density, but if we take it to be too low then the scoring function takes longer (since the search algorithm has to go through more nodes for each maze) meaning slower progress at the beginning, computation wise. If we take it to be too dense it takes more generations before mazes will have a path to the end and it can start optimising the length of the path, also resulting in slower progress at the beginning, but this time because it needs more generations to get to the same result. Again we find 0.7 to work fine as a begin density.

The tuning of parameters, scoring functions and methods seems to be an endless process.



```
Best maze of generation 1000 with score
 185.49999999999918:
= = = = = = = = = = = = = = = = = = = = = = = = =
=           |                   |               =
= - +   + - + - +   + - + - +   +   + - + - +   =
=       |           |       |       |           =
=   + - +   + - + - +   +   + - + - + - +   + - =
=       |           |   |   |               |   =
=   + - + - + - +   +   +   + - +   + - + - +   =
=   |               |   |   |                   =
=   + - +   + - + - +   +   + - + - + - +   + - =
=       |           |   |   |           |       =
= - +   +   +   + - +   +   +   + - +   + - +   =
=       |   |   |   |   |       |           |   =
=   + - + - +   +   +   + - + - +   + - +   +   =
=   |           |   |   |       |   |           =
=   +   +   + - +   +   +   +   + - + - + - + - =
=   |   |   |           |   |           |       =
=   +   + - +   + - +   +   + - +   + - +   +   =
=       |           |   |   |               |   =
= - + - +   + - + - + - +   +   +   + - + - +   =
=       |       |           |   |   |           =
=   +   +   + - +   +   + - + - + - +   + - +   =
=   |       |   |   |   |               |   |   =
=   + - + - +   +   + - +   + - + - + - +   + - =
=                       |                       =
= = = = = = = = = = = = = = = = = = = = = = = = =
```

Figure 3. Evolutionary algorithm: The resulting maze from 1000 generations of mazes, each consisting of 120 mazes