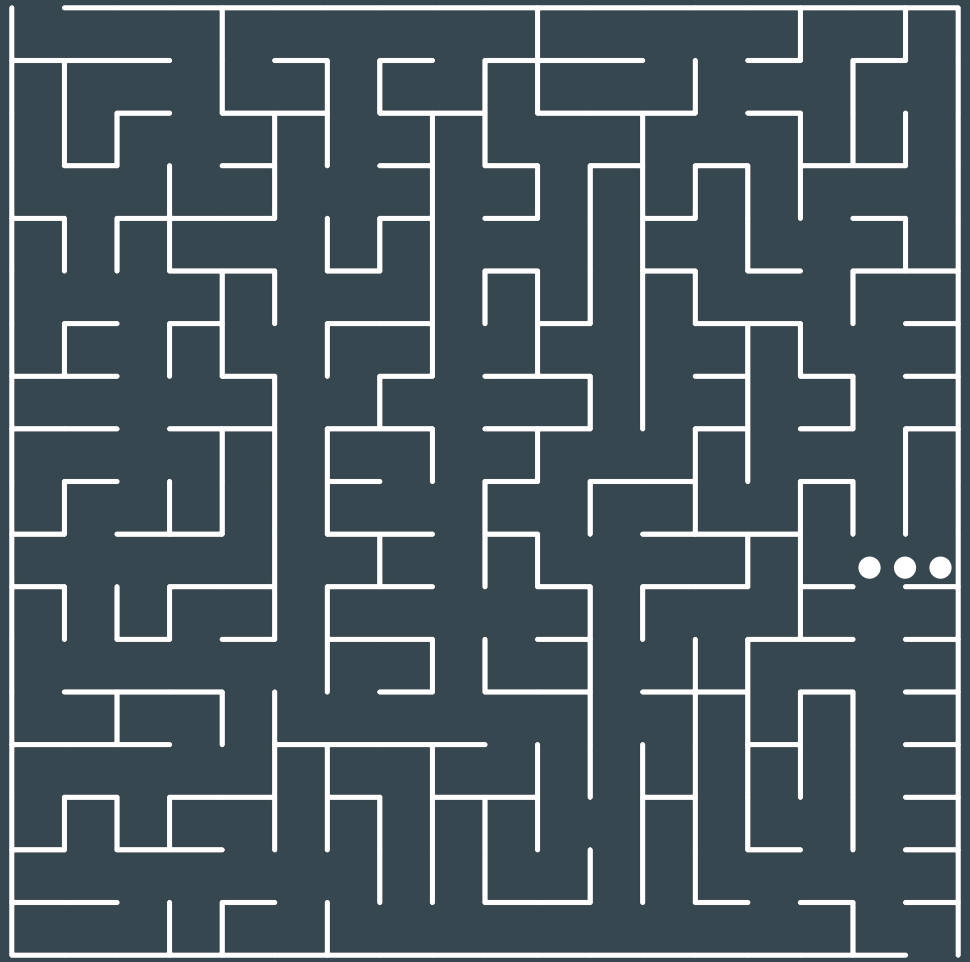

code cademy/PRO intensive
capstone project

Convoluted Kernel Maze

What would the maze look like if we'd moved more than 2 cells?

The walls of the maze would just be thicker. When moving only 2 cells, we are making sure, that a wall is only one cell wide. If we'd move 3 cells a wall would be 2 cells wide and so forth. A 20x20 maze, that was constructed using the 3 cell method, would have "less" path and would look more sparse, than a maze that was constructed using the 2 cell method.

What would the maze look like if we'd move a random amount of cells each time?

If we'd move a random amount of cells at each iteration of the mowing loop, we'd end up with a maze that would have thicker walls as well as thinner walls and when moving only 1 cell, the maze could even have "rooms", where a empty cell has more then two empty neighbors. Probably, the best method for your generic AD&D session. A dragon won't fit into that narrow corridors...

What algorithm could you use to find a path through this maze? And how does knowing the generation algorithm influence the decision?

If we treat each empty cell as a independent node, and therefore treat the path in the maze as an unweighted graph(see implementation in code), we can use both: DFS (depth first search) and BFS (breadth first search), resulting in a run time of O(vertices+edges). We could reduce the number of nodes, if we only considered each dead end, turn of direction or forking to be an actual node. This would not be enough though to result in a different Big O.

Since we know that there is definitely only one solution to this maze, we do not need an algorithm that gives us the shortest path. Dijkstra's would only be useful if we'd extract a weighted graph, where the start and the end node are connected by more than just one path. An A* algorithm might not be suitable at all, since the coordinates of the start and end point might be pretty close together, sending the the A* algorithm in a totally wrong direction. This is a maze, after all. It is supposed to be "misleading".

How might you best store the list of items you've collected?

In order to obtain as much information about the swag items and have plenty of options for sorting, it might be best/easiest to store them in a list of lists, where each nested list contains the coordinates (in form of a tuple or a cell name) where the specific item was found and then the item itself. In my implementation I would opt for the cell name. Or a total refactoring of make_graph()... ;)

Example using tuples:

swag_list = [[(3, 12), 'werewolf], [(5, 5), 'pumpkin'], [(10, 14), 'candy corn']]

Example using cell names (i.e. numbers)

swag_list = [[101, 'werewolf], [56, 'pumpkin'], [204, 'candy corn']]

What sorting algorithms would you consider using (assume a much larger list of possible swag? And how does variety of swag influence your answer.

If the swag items only consisted of integers the best option for sorting might be a radix sort, since this type of sort provides a good time efficiency of O(n). Even really large item lists could be handled quite fast this way.

If the swag item consist of string or mixed data types Quick sort would provide an average running time complexity of O(n log(n)).