Convoluted Kernel Maze

Nicolai Banke

Codecademy

2019

Maze Solving

- A solution to this maze is first and foremost a sequence of steps from begging to end
- One could choose to find the shortest path, but instead I will attempt to find the path that allows for the largest amount of swag
- To this end, I will write a function that represents the maze as a *graph*

Maze Solving

- Each point in the graph that is not a 'wall' represents a vertex in the graph, and each point (i, j) is connected to those its immediate neighbours $(i \pm 1, j \pm 1)$ that don't represent a 'wall'
- Furthermore, the graph will be made directed by only being connected to its neighbour in the forward direction, defined by the location of the starting point

Maze Solving

A few things to consider before choosing a search algorithm:

- Depth First Search is used when we want to check if there is a path through the graph, but won't necessarily find the shortest one
- Breadth First Search is used when we want to find the shortest path through a graph, but is inefficient if there is no solution

Maze Solving

- When mowing through the maze during its construction, fewer moves will result in more intersections and blind alleys in the maze, and thus more ways that a patron can turn and get lost
- With more moves the maze will end up having longer straight paths and fewer ways that a patron can get lost
- In the latter case it might be better to use a DFS algorithm, since with fewer ways to turn and end up in a blind alley, any one path would be more likely to be the one reaching the end

Maze Solving

- In the former case, it is, for the opposite reasons, then better to use BFS
- Going through this maze, a patron also want to pick up some swag along the way, and would want to pick a route that contains the best swag and avoids the undesirable swag
- Furthermore, since the swag was dropped randomly by using a BFS queue in the *explore*-function, a BFS approach to finding the swag is more appropriate

Maze Solving

- Each swag item is then mapped to a value with desirable swag having negative value and undesirable swag having positive value
- Solving the maze in this way calls for Dijkstra's algorithm, where the graph vertices are as described before and the values of the different swag items are edge weights
- Dijkstra's algorithm is then straight forward to implement and will return the path with least "distance", i.e. the path collecting most desirable swag

Maze Solving

- Another way to solve the maze would be to find the path with the most desirable swag, while still trying to walk the shortest route possible
- This can be solved using the A^* algorithm using a heuristic, and these two algorithm have been implemented

Sorting Algorithms

Things to consider before choosing a sorting algorithm for the collected swag

- We have a large amount of mostly unsorted items since random swag has been tossed and picked up at random
- *Bubblesort* has runtime $O(N^2)$ and will only be quick for mostly sorted items
- *Quicksort* has worst-case runtime of $O(N^2)$ and average run-time $O(N \log N)$
- *Mergesort* has run-time $O(N \log N)$ even in the worst case, so we will pick this sorting algorithm

Conclusions and Final Thoughts

- A modified version of Dijkstra's algorithms seems to have been implemented correctly, in that it returns a list of points representing the path, and a list of collected swag, sorted according to mergesort
- The A^* algorithm returns a path and list of swag as well, but I noted that while A^* never collected more swag than Dijkstra's as expected, the two algorithms always found the same path
- I haven't been able to figure out why that is, or if it is to be expected

Conclusions and Final Thoughts

- I tried to change the frequency with which swag was dropped, the size of the maze and the order of magnitude of the swag values in an attempt to make the effect of the heuristic against the 'edge weights' more clear, but it didn't seem to make much of a difference
- Perhaps if the swag wasn't uniformly distributed in the maze, but rather, was concentrated around the corner of the maze opposite the ending point, we would see the A^* algorithm favour a path with fewer points and much less swag than the one found by Dijkstra's