ECM2423 – Coursework exercise

Question 1.1

By definition, a maze if defined by a set of paths which lead to different points in the maze. It has an entry point on one end, and an exit point, which we will assume to be at the other end for this exercise (in some cases, that goal destination is the “centre” of the maze).

A maze can be seen as a graph: we can consider each point in the maze (given by a set of coordinates) to be a node, and a path between adjacent points (i.e., not a wall) as an edge between two nodes. To solve a maze, one attempts to find a path between the entry point and the exit point: this can be formulated as a search problem with a state space, a start, and a goal condition defined below.

The state space of the problem is the set of all paths within the maze. The start is assumed to be always the same: it is the entry point of the maze with coordinates (0,1). The goal condition is reaching the exit point of the maze, which is situated on the other side of the maze from the entry point (for this exercise, this is somewhere along the lowest ‘points’ of the 2D maze).

We can also define the problem situations/states as the different points (represented by coordinates) where one can navigate through the maze, i.e., the points which are not walls. The possible actions/operators can be given by moving to an adjacent point (not blocked by a wall) below, above or to the left or right of the current point. Finally, the length of the path from the start to the goal can be seen as the path cost of the problem.

Note that it would be possible to consider that the start of the maze can be anywhere along the topmost ‘points’ of the maze, but for simplicity we will initially assume that the start point is always at (0,1).

Question 1.2

1. The depth-first algorithm

The depth-first algorithm is used to traverse or search a graph from some root node, exploring as far as possible along each branch before backtracking. The algorithm uses a stack to keep track of nodes on a specific branch, therefore allowing it to explore nodes in ‘deeper levels’ of a branch following the Last-In-First-Out (LIFO) principle of the stack. In the context of this exercise, we will be using the depth-first approach to search the graph (rather than fully traversing it), aiming to find a path from the start to the goal of the maze.

Depth-first search has several properties that we need to keep in mind. Firstly, it does not guarantee completeness, i.e., it is susceptible to infinite loops. This means we will need to check for cycles by keeping track of which nodes we have explored. The time complexity of the algorithm is given by $O(|V| + |E|)$, where $|V|$ is the number of vertices/nodes in the graph and $|E|$ is the number of edges. This is quite high, since in the worst case, the algorithm would visit all nodes in the graph.

Depth-first search needs to remember the vertices on the current path as well as the ones already visited, meaning its space complexity in the worst case is $O(|V|)$, which is relatively low. Finally, it does not guarantee optimality: the path given by the algorithm may not be the shortest solution, as it does not take cost (path length) into consideration.

2. Implementing depth-first search to solve a maze

The easy maze provided in the specification (through the file 'maze-Easy.txt') can be solved by executing the main program and entering 's' to solve the sample mazes (see README). The path found by the depth-first search algorithm is given by the following set of coordinates in order from the start node (0, 1) to the goal node (9, 18):

(0,1), (1,1), (1,2), (1,3), (1,4), (1,5), (2,5), (3,5), (4,5), (5,5), (5,6), (5,7), (5,8), (6,8), (6,9), (6,10), (6,11), (6,12), (6,13), (6,14), (6,15), (6,16), (6,17), (7,17), (8,17), (8,18), (9,18)

3. Performance statistics for 'maze-Easy.txt'

Full performance statistics for solving the easy maze using depth-first can be found in the output file produced when solving all sample mazes or this maze only (see README):

- Path length: 27
- Explored nodes: 45
- Steps: 66
- Execution Time: 4.7e-05 s

Overall, these are some promising results, but they are not the most relevant as the maze is relatively simple. We will be covering statistics for larger mazes in the next section.

Note that the execution time listed above (and similarly for all mazes and algorithms) includes the time spent finding the goal node of the maze in addition to the actual execution time of the algorithm. This is done under the assumption that finding the goal node is theoretically part of the process of finding the path. It requires some form of computation, since its coordinates are unknown before solving the maze, as opposed to the starting point which is always assumed to be at (0,1).

4. Algorithm generalisation and performance statistics for all mazes

All mazes provided as part of the CA can be solved using depth-first search through the main program (see README). Full paths and statistics can be found in an output file called 'sample-results.txt' located in the same folder as the main script.

The main program also provides the ability to read any maze in the same format by entering 'c' and then the name of the input file, provided it is in the same folder as 'main.py'.

Below are the statistics obtained by solving the example mazes using depth-first search. Please note that the paths found are not listed in this report and can instead be found in the output file.

- 'maze-Medium.txt': the algorithm explored a total of 2493 nodes, finding a path of length 339 in 4693 steps, and taking around 5.4e-03 s or 5.4 ms.

- For ‘maze-Large.txt’ the algorithm explored a total of 10308 nodes, finding a path of length 1092 in 19664 steps, and taking around 4.9e-02 s or 49 ms.
- For ‘maze-VLarge.txt’, the algorithm explored a total of 113322 nodes, finding a path of length 3737 in 223554 steps, and taking around 4.2 s.

Question 1.3

1. Choosing an improved algorithm for this problem

While our implementation of depth-first does seem quite efficient, it might not perform as well for much larger mazes and can certainly be improved. Moreover, depth-first search does not guarantee an optimal solution, meaning that the path found by the algorithm may not be (and probably will not be for most cases) the shortest path from the entry point of the maze to the exit point. Finding the shortest path is desirable in many situations, so we choose an algorithm that can do this: A* graph search.

Unlike depth-first, it uses an informed or heuristic approach: this means that the algorithm uses context specific knowledge (in this case, the length of a path) to pick which node to expand. For A*, this is done through an evaluation function $f(n)$, given by the sum $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of reaching the node n, and $h(n)$ is the heuristic function estimating the cost of reaching the goal node from the node n. Hence $f(n)$ gives the estimated cost of the cheapest path to the goal node that goes through the node n.

If $h(n)$ is admissible, i.e., if it always underestimates the cost of reaching a goal, A* graph search gives the optimal solution, which in our case is the shortest path from the start to the goal. This is exactly what we are looking for, along with the hope of reducing execution time for larger mazes.

For smaller mazes, we acknowledge that A* may be slower than depth-first search, which does not have the overhead associated with computing the heuristic. However, we are mostly looking to improve performance on large mazes, where the difference in execution time is likely to be larger.

2. Implementation of A* graph search

For our implementation of A* graph search to find the shortest path, we will need to choose the optimal heuristic function for this problem. For solving a maze with only the four cardinal directions, the optimal heuristic is theoretically the Manhattan Distance, which is the horizontal plus vertical distance between two points. For two points $p_1(x_1, y_1)$ and $p_2(x_1, y_1)$, this is given by:

$$h(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

The easy maze provided in the specification (‘maze-Easy.txt’) can be solved by executing the main program. The path found by the A* graph search algorithm is given by the following set of coordinates in order from the start node (0, 1) to the goal node (9, 18):

(0,1), (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (1,7), (1,8), (1,9), (1,10), (1,11), (1,12), (1,13), (1,14), (1,15), (2,15), (3,15), (4,15), (5,15), (6,15), (6,16), (6,17), (7,17), (8,17), (8,18), (9,18)

Here are the performance statistics for this maze:

- Path length: 27
- Explored Nodes: 41
- Steps: 61
- Execution Time: 2.7e-04 s

The path found by A* is different than for depth-first, despite having the same length. Both algorithms are similar in terms of performance, with A* being slightly slower, as was expected for small mazes.

3. Algorithm generalisation and performance statistics for all mazes

As for depth-first, all mazes provided in the specification can be solved using A* graph search by running the main program. The obtained paths and statistics can then be found in the output file ‘sample-results.txt’. Similarly, any maze in the same format can also be read by entering ‘c’ and then the name of the input file.

Below are the statistics obtained by solving the example graphs using A* graph search. The paths found are not listed here and can instead be found in the output file.

- ‘maze-Medium.txt’: the algorithm explored a total of 2009 nodes, finding a path of length 321 in 2271 steps, and taking around 1.4e-02 s or 14 ms.
- ‘maze-Large.txt’: the algorithm explored a total of 41623 nodes, finding a path of length 974 in 42364 steps, and taking around 0.25 s or 250 ms.
- ‘maze-VLarge.txt’: the algorithm explored a total of 273474 nodes, finding a path of length 3691 in 276715 steps, and taking around 2.3 s.

4. Performance analysis and comparison

Overall, the results provided by the A* graph search algorithm are relatively aligned with our previous assumptions. Firstly, the final paths found by the algorithm are shorter than for depth-first search for all example mazes, except ‘maze-Easy.txt’ for which it is equal. This was to be expected, as our implementation of A* should give the shortest path, whereas depth-first does not guarantee an optimal solution.

Finding the shortest path may come at a cost, though. For ‘maze-Easy.txt’ and ‘maze-Medium.txt’, this is mostly seen through the slightly longer execution time of A* (although this is not exactly relevant considering how short it already is). For both these mazes, the number of explored nodes, as well as the number of steps involved, is still relatively similar.

The difference in performance is much more visible for ‘maze-Large.txt’ and especially ‘maze-VLarge.txt’. For both these mazes respectfully, the number nodes explored by A* is four and twice as much as depth-first. The number of steps involved is also higher for A* than for depth-first, although the difference does seem to be smaller for increasingly larger mazes.

While depth-first seems to outperform A* in terms of execution time for the first three mazes, this is certainly not the case for ‘maze-VLarge.txt’. In fact, A* is around two seconds faster than depth-first for this maze, almost halving its execution time. This is also what we expected: the informed approach used by A* seems to be paying off in increasing the efficiency of the algorithm for larger mazes.

The number of steps taken by both algorithms to solve the largest maze is also quite similar, although A* explores many more nodes than depth-first. By definition, depth-first explores the individual branches of the maze as far as possible, meaning it may have to backtrack a lot if a deep branch is found to be a dead-end. This might be what causes it to have such a large number of steps despite a lower explored node count, and could be responsible for the performance deficit compared to A*. The latter has almost as many explored nodes as it has steps, and it takes as many steps as depth-first to explore more than twice as many nodes, and doing it all faster than its counterpart. If it does not affect execution time too much, exploring more nodes could be good because it can give a better picture of the entire maze.

Overall, we can consider the use of A* graph search to be a success for maze solving, especially when compared to depth-first search. Not only does it find the shortest possible path unlike depth-first, but its execution time is also significantly faster for larger mazes. In the next section of this report, we will be experimenting further by modifying both algorithms to compare performance.

Further experimentation

1. Depth-first search - ordering the expansion of neighbouring nodes

With depth-first search, the order in which the neighbours of a given node are explored is determined by their order in the stack. The mazes provided as part of the specification generally followed the same tendencies for start and goal positions: the starting node was at the top left corner of the maze, and the goal node was at the lowest level, usually near the bottom right corner. Hence in our initial implementation, the neighbours of a given node were expanded in what seemed like the most intuitive order to reach the goal node: first the node under the current node, then the one to its right, followed by the one to its left, and finally the one above it.

In an effort to improve our initial implementation, we experimented with different orderings in the expansion of neighbouring nodes, and compared the performance statistics for all mazes. We will only be covering 'maze-VLarge.txt' in this report, but the performance details for all other mazes can be found in the file 'stats_dfs_orderings.txt'. Here were the statistics for 'maze-VLarge.txt':

- Expansion order 1 : bottom - right - left – top
    - Path length: 3737
    - Explored nodes: 113322
    - Steps: 223554
    - Execution Time: 4.23 seconds

- Expansion order 2: right - bottom - left – top
    - Path length: 3737
    - Explored nodes: 380656
    - Steps: 758039
    - Execution Time: 18.48 seconds

- Expansion order 3: left - bottom - right – top
    - Path length: 4053
    - Explored nodes: 88662

  - Steps: 175799
  - Execution Time: 2.44 seconds

- Expansion order 4: left - right - bottom – top
  - Path length: 4049
  - Explored nodes: 134772
  - Steps: 267881
  - Execution Time: 5.98 seconds

Note that while there are $4! = 24$ ways of ordering the neighbours of a given node, we have focused on only four of them for this report, including the one used in the initial implementation of the algorithm.

The second ordering seems to find a path with the same length as the first one, but the other orderings find different paths which are both longer. However, the second ordering has more than three times as many explored nodes and steps as the first one, which leads to the much longer execution time of over 18 seconds. The third ordering performs really well for this maze, completing in only 2.44 seconds against 4.2 seconds for the first ordering. Despite a slightly longer final path, it boasts a significantly lower explored node count, as well as a lower step count. Finally, the fourth ordering displays a similar performance to the first one, with an execution time of around 6 seconds.

How valuable are these results? From these statistics, it appears that the third ordering would be better suited than the first one for this maze. However, considering we have not experimented with the 20 other possible orderings, it is also possible that one of them would beat this one as well. Realistically, these results are only relevant for this particular maze: for a maze with a completely different layout, any ordering might be the best. Therefore, these results don't actually help us improve the efficiency of our depth-first search implementation for general maze solving, but rather only for solving 'maze-VLarge.txt'.

However, since maze layouts are generally somewhat similar (e.g., with a starting point in the top left corner and an exit point near the bottom right), there is still probably some tendency for certain orderings to perform better than others. It would be interesting to build a model to calculate similar statistics for all 24 possible orderings, and feed it with a vast number of mazes with larger and more varied layouts. While even this would probably not provide us with the "optimal" ordering of neighbour nodes expansion for depth-first, it would certainly give us a better idea of which orderings generally perform well.

2. A* graph search – experimenting with the heuristic

In A* graph search, the order in which the neighbours of a given node are expanded depends on the output of the heuristic function which estimates the cost of reaching the goal node. Hence, we cannot experiment with fixed orderings like we did for depth-first search. Instead, we can experiment by modifying the heuristic function $h(n)$.

Our implementation of A* already gives the shortest path from the start to the goal node in the maze. It uses the Manhattan Distance as a heuristic function, which is admissible in this case. This is because the only possible operations for a maze are to move in the four cardinal directions, so the Manhattan Distance is theoretically the shortest distance between two points.

Considering the algorithm is already optimal, we can focus on assessing the performance of our current heuristic and comparing it with other functions. To do this, we can start by setting $h(n) = 0$ for all n. This essentially gives the 'uninformed version' of A* graph search, which is Dijkstra's algorithm.

It will give us some form of baseline to compare our current and other heuristics with. Note that it can be run on any maze from the main program (see README). Here are the performance statistics obtained when solving 'maze-VLarge.txt' using Dijkstra's algorithm:

- Path length: 3691
- Explored nodes: 796950
- Steps: 800266
- Execution Time: 5.68 seconds

The path found is the same as the one found by A* since Dijkstra's algorithm is also optimal: its heuristic $h(n) = 0$ is admissible because for each node n, we have $(n) \leq h*(n)$, where $h*(n)$ is the minimum cost to reach the goal (which cannot be smaller than 0). Dijkstra's algorithm explores almost three times as many nodes and completes more than twice as steps as A*, in over twice its execution time. Hence it does seem that our initial heuristic for A* guides the search rather effectively and performs well compared to Dijkstra's algorithm.

Forgetting the idea of needing an optimal solution, we will now be experimenting with another heuristic to see if we can reduce the execution time of the algorithm. Adding a weight to the Manhattan Distance gives the following heuristic:

$$h(p_1, p_2) = 2 * (|x_1 - x_2| + |y_1 - y_2|)$$

An algorithm using this kind of weighted heuristic is called 'weighted A*' and does not guarantee an optimal solution. Here are its performance statistics when solving 'maze-VLarge.txt':

- Path length: 3691
- Explored nodes: 42528
- Steps: 48929
- Execution Time: 1.02 seconds

Interestingly, the path found by weighted A* has the same length as both Dijkstra's algorithm and our initial implementation of A*, despite not being optimal. It explored less than a sixth of the nodes explored by the initial A* and divided the number of steps similarly. It also took half as much time to execute. Although it does not guarantee an optimal solution, weighted A* does seem to significantly improve the performance of our initial implementation of A*.

We even managed to get an execution time of just under one second (0.9977 s) for the same path by tweaking the weight from 2 to 2.7, although this value altered between 0.99 and 1.01 when running the algorithm multiple times. Overall, our goal of assessing and improving the performance of our initial heuristic has been achieved.